INTERFAZ GRÁFICA Y FASE DE PRUEBAS DE *SOFTWARE* DE VÍDEO FORENSE

(SISTEMA DE CODIFICACIÓN DV)

PFC: PEDRO M. MATAMALA LUCAS

Ingeniería Técnica de Telecomunicación, Sonido e Imagen

EUITT, Universidad Politécnica de Madrid

2012/2013

PROYECTO FIN DE CARRERA

PLAN 2000

E.U.I.T. TELECOMUNICACIÓN

TEMA: fase final de desarrollo de la aplicación de vídeo forense SAVID

TÍTULO: Interfaz Gráfica y Fase de Pruebas, de software de vídeo forense (Sistema de codificación DV).

AUTOR: Pedro Miguel Matamala Lucas

TUTOR: Luis Ignacio Ortiz Berenguer **Vº Bº.**

DEPARTAMENTO: DIAC

Miembros del Tribunal Calificador:

PRESIDENTE: D. Matías Garrido González

VOCAL: D. Luis Ignacio Ortiz Berenguer

VOCAL SECRETARIO: D.ª Elena Blanco Martín

DIRECTOR: D. César Sanz Álvaro

Fecha de lectura:

Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

El presente proyecto fin de carrera, realizado por el ingeniero técnico en telecomunicaciones Pedro M. Matamala Lucas, es la fase final de desarrollo de un proyecto de mayor magnitud correspondiente al software de vídeo forense SAVID. El propósito del proyecto en su totalidad es la creación de una herramienta informática capacitada para realizar el análisis de ficheros de vídeo, codificados y comprimidos por el sistema DV –*Digital Video*-. El objetivo del análisis, es aportar información acerca de si la cinta magnética presenta indicios de haber sido manipulada con una edición posterior a su grabación original, además de mostrar al usuario otros datos de interés como las especificaciones técnicas de la señal de vídeo y audio. Por lo tanto se facilitará al usuario, analista de vídeo forense, información que le ayude a valorar la originalidad del contenido del soporte que es sujeto del análisis.

RESUMEN

El presente proyecto fin de carrera, realizado por el ingeniero técnico en telecomunicaciones Pedro M. Matamala Lucas, es la fase final de desarrollo de un proyecto de mayor magnitud correspondiente al software de vídeo forense SAVID. El propósito del proyecto en su totalidad es la creación de una herramienta informática capacitada para realizar el análisis de ficheros de vídeo, codificados y comprimidos por el sistema DV –*Digital Video*-. El objetivo del análisis, es aportar información acerca de si la cinta magnética presenta indicios de haber sido manipulada con una edición posterior a su grabación original, además, de mostrar al usuario otros datos de interés como las especificaciones técnicas de la señal de vídeo y audio. Por lo tanto, se facilitará al usuario, analista de vídeo forense, información que le ayude a valorar la originalidad del contenido del soporte que es sujeto del análisis.

El objetivo específico de esta fase final, es la creación de la interfaz de usuario del software, que informa tanto del código binario de los sectores significativos, como de su interpretación tras el análisis. También permitirá al usuario el reporte de los resultados, además de otras funcionalidades que le permitan la navegación por los sectores del código que han sido modificados como efecto colateral de la edición de la cinta magnética original.

Otro objetivo importante del proyecto ha sido la investigación de metodologías y técnicas de desarrollo de software para su posterior implementación, buscando con esto, una mayor eficiencia en la gestión del tiempo y una mayor calidad de software con el fin de garantizar su evolución y sostenibilidad en el futuro. Se ha hecho hincapié en las metodologías ágiles que han ido ganando relevancia en el sector de las tecnologías de la información en las últimas décadas, sustituyendo a metodologías clásicas como el desarrollo en cascada. Su flexibilidad durante el ciclo de vida del software, permite obtener mejores resultados cuando las especificaciones no están del todo definidas, ajustándose de este modo a las condiciones del proyecto.

Resumiendo las especificaciones técnicas del software, C++ es el lenguaje de programación orientado a objetos con el que se ha desarrollado, utilizándose la tecnología MFC -*Microsoft Foundation Classes*- para la implementación. Es un proyecto MFC de tipo cuadro de dialogo,

creado, compilado y publicado, con la herramienta de desarrollo integrado Microsoft Visual Studio 2010. La arquitectura con la que se ha estructurado es la arquetípica de tres capas, compuesta por la interfaz de usuario, capa de negocio y capa de acceso a datos. Se ha visto necesario configurar el proyecto con compatibilidad con CLR –*Common Languages Runtime*- para poder implementar la funcionalidad de creación de reportes.

Acompañando a la aplicación informática, se presenta la memoria del proyecto y sus anexos correspondientes a los documentos EDRF –Especificaciones Detalladas de Requisitos funcionales-, EIU –Especificaciones de Interfaz de Usuario , DT -Diseño Técnico- y Guía de Usuario.

SUMMARY

This dissertation, carried out by the telecommunications engineer Pedro M. Matamala Lucas, is in its final stage and is part of a larger project for the software of forensic video called SAVID. The purpose of the entire project is the creation of a software tool capable of analyzing video files that are coded and compressed by the DV -Digital Video- System. The objective of the analysis is to provide information on whether the magnetic tape shows signs of having been tampered with after the editing of the original recording, and also to show the user other relevant data and technical specifications of the video signal and audio. Therefore the user, forensic video analyst, will have information to help assess the originality of the content of the media that is subject to analysis.

The specific objective of this final phase is the creation of the user interface of the software that provides information about the binary code of the significant sectors and also its interpretation after analysis. It will also allow the user to report the results, and other features that will allow browsing through the sections of the code that have been modified as a secondary effect of the original magnetic tape being tampered.

Another important objective of the project is the investigation of methodologies and software development techniques to be used in deployment, with the aim of greater efficiency in time management and enhanced software quality in order to ensure its

development and maintenance in the future. Agile methodologies, which have become important in the field of information technology in recent decades, have been used during the execution of the project, replacing classical methodologies such as Waterfall Development. The flexibility, as the result of using by agile methodologies, during the software life cycle, produces better results when the specifications are not fully defined, thus conforming to the initial conditions of the project.

Summarizing the software technical specifications, C + + the programming language – which is object oriented and has been developed using technology MFC- Microsoft Foundation Classes for implementation. It is a project type dialog box, created, compiled and released with the integrated development tool Microsoft Visual Studio 2010. The architecture is structured in three layers: the user interface, business layer and data access layer. It has been necessary to configure the project with the support CLR -Common Languages Runtime – in order to implement the reporting functionality.

The software application is submitted with the project report and its annexes to the following documents: Functional Requirements Specifications - Detailed User Interface Specifications, Technical Design and User Guide.

CONTENIDO

1. OBJETIVOS DEL PROYECTO

En el estándar de vídeo DV, *Digital Video,* se especifica cómo en ciertos sectores del código almacenado en una cinta magnética o soporte no convencional se registran trazas y se etiquetan ciertos bits cuando, sobre el soporte original, se realiza una edición. Por tanto, si la información original capturada por una videocámara que codifica y comprime la imagen y el audio bajo esta norma es manipulada, tras un análisis exhaustivo del código podría comprobarse la autenticidad de su contenido.

El proyecto, iniciado por el doctor Luis I. Ortiz Berenguer, tiene como objetivo el desarrollo de una herramienta de *software* de vídeo forense capacitada para realizar independientemente la demodulación, la decodificación y el análisis de dos fotogramas comprimidos en el sistema DV. El objetivo final del análisis es identificar e interpretar las trazas registradas en el código para informar al usuario de una hipotética manipulación.

El proyecto de fin de carrera, llevado a cabo por el ingeniero técnico Pedro M. Matamala Lucas, se corresponde con la fase final de todo el proyecto. La versión del *software* SAVID de la que se parte archiva el resultado de la decodificación en diferentes documentos binarios en función del tipo de información, sin que se realice ningún análisis del contenido.

Los objetivos específicos de esta última fase son los siguientes:

- Integración con el desarrollo previo
- Análisis de los datos decodificados de los dos fotogramas, identificando los sectores significativos
- Presentación de resultados mediante IU (Interfaz de Usuario)
- Capacitar al usuario de funcionalidad en gestión, administración, almacenamiento y envío de informes en diferentes formatos
- Fase de pruebas

- Redactar la documentación que permita el mantenimiento futuro de la aplicación

Para realizar la integración con la versión previa de SAVID, se analizará la aplicación para poder describir diferentes escenarios y así poder elegir justificadamente el más adecuado a seguir en el nuevo desarrollo. Para esto se tendrán en cuenta los posibles problemas de compatibilidad entre la IDE, Herramienta de Desarrollo Integrado, con la que se programó la solución actualmente obsoleta y los beneficios que ofrecen las últimas versiones del IDE con nuevas funcionalidades.

El análisis del contenido de los archivos requiere un estudio previo de la norma, haciendo hincapié en los sectores que aportan información útil a la hora de autentificar la originalidad del contenido. Se estudiará la forma con la que la nueva aplicación obtendrá la información de los archivos binarios y qué especificaciones técnicas tendrá la capa de negocio para evaluar los resultados.

Al usuario se le mostrará mediante IU tanto la información general de las características técnicas del contenido del archivo de vídeo como los resultados del análisis de los sectores significativos. Se mostrará la captura de ambos fotogramas y la información del contenido de audio.

El usuario podrá reportar los resultados en diferentes formatos, permitiéndole el archivo y la impresión de los informes. Se estudiará qué formatos son los más adecuados y cómo se implementan en la tecnología de desarrollo de las acciones comentadas.

En un principio, la fase de pruebas no solo englobaba las consideradas puramente de *software*, como son las pruebas de verificación, sino que se pretendía realizarlas con un gran número de archivos y así poder cubrir también las pruebas de validación de usuario. La lentitud para adquirir este volumen de archivos de los mecanismo burocráticos ajenos a nuestra institución plantean otro escenario diferente, por lo que se documentará un plan de pruebas que sirva para realizar, en un futuro, esta fase que permita la validación de la solución.

Toda la documentación que se aporta en el presente proyecto incluye tanto la que da soporte al usuario como la que permitirá a desarrolladores realizar futuros evolutivos sobre la aplicación.

El proyecto no solo pretende desarrollar un *software* para dar solución a la funcionalidad descrita, sino que también desea profundizar en metodologías y técnicas que se aplican actualmente en el sector IT, por lo que se tratará de seguir con criterio las siguientes pautas:

- Garantizar la calidad del código
- Arquitectura justificada de *software*
- Metodologías de trabajo y desarrollo que mejoren la productividad y la gestión del tiempo
- Orientarlo al ámbito académico aplicando técnicas del mundo empresarial

2. INTRODUCCIÓN AL SISTEMA DV

Antes de profundizar en los aspectos que conciernen a los objetivos principales del presente proyecto, se ha visto conveniente hacer una introducción a la tecnología de vídeo que será objeto de análisis. Aspectos físicos como son los soportes de almacenamiento, la magnetización, la modulación y otros aspectos técnicos como la comprensión, al centrarse en el análisis de los datos decodificados, no se abordarán en este proyecto, como se ha descrito anteriormente —concretamente, en el análisis de los sectores de código que contienen la información útil para dar solución a la funcionalidad planteada.

Este apartado comienza describiendo, técnicamente, el estándar original DV y sus variantes comerciales, especificadas por los organismos de normalización y comercializadas por los principales fabricantes de videocámaras, magnetoscopios y reproductores. Para finalizar, se plantea cuál es la situación actual del formato y las previsiones en un futuro próximo, haciendo hincapié en su adaptación a soportes no convencionales de almacenamiento y adquisición de datos.

2.1 ESTÁNDAR DV

El estándar de vídeo DV fue definido en 1995 por los principales fabricantes de videocámaras en un consorcio de unas 60 compañías entre las que destacaban Sony, Panasonic, JVC e Hitachi, siendo posteriormente especificado en la norma IEC 61834. Los formatos basados en DV, el DVCPRO y el DVCAM, fueron especificados en la norma SMPTE 314M y el formato de alta definición DVCPRO HD en la norma SMPTE 370M. En su inicios fue desarrollado para soportes de almacenamiento en cinta magnética de cuarto de pulgada (6.3 mm) en los tamaños de casete *Mini*, *Medium* y *Large*. En la actualidad, han proliferado nuevos soportes que permiten mayores tasas de escritura, mayor capacidad de almacenamiento y han evolucionado de conceptos como la captura a otros como el volcado de datos, con lo que esto

implica; por tanto, estos formatos se han adaptado para ser soportados por discos duros, memorias de estado sólido o soportes ópticos.

Aunque fue pensado como formato de vídeo para el ámbito industrial, al igual que otros formatos audiovisuales, pasó a comercializarse como formato de vídeo doméstico, principalmente, por sus altas prestaciones a bajo coste, revolucionando el mercado al introducir los sistemas de vídeo digital en este ámbito.

Ilustración 1: variantes sistema DV

Respecto a sus características técnicas, realiza una compresión con pérdidas para la señal de vídeo y sin pérdidas para las señales de audio. Es un sistema de vídeo digital por componentes YCbCr, con submuestreo de crominancia 4:2:0 y 4:1:1 en PAL y 4:2:0 en NTSC, siendo la frecuencia de muestreo del componente de luminancia "Y" de 13,5 MHz. Se basa en un algoritmo DCT (Transformada de coseno discreta) para la compresión y, tras el análisis frecuencial, realiza una codificación basada en la técnica *Intra-frame* de 5:1 de radio para explotar la redundancia espacial, dando un flujo resultante de 25 Mb/s. El audio utiliza el procedimiento de modulación digital PCM (Modulación por impulsos codificados).

El protocolo de conectividad digital del estándar es, por excelencia, el IEEE 1394, *firewire*, en sus versiones de 6 pines con alimentación o 4 pines sin alimentación. Muchas videocámaras también ofrecen conectividad por USB 2.0, permitiendo la transferencia de máxima calidad DV solo en aquellas videocámaras que usan el controlador UVC; el resto, en caso de ofrecer

esta alternativa, utilizan esta interfaz para transferir imágenes fijas. Algunas cámaras profesionales permiten conectividad por SDI (*Serial Digital Interface*) y SDTI (*Serial Digital Transport Interface*), desarrollada para satisfacer la transmisión de vídeo comprimido, usando en ambos casos el conector BNC. Por último, algunos dispositivos permiten conectividad por medio de la interfaz analógica de vídeo por componentes, con las correspondientes pérdidas debido a la conversión analógico-digital.

Ilustración 2: interfaces sistema DV

Las tramas de datos de un flujo en el formato DV se dividen en varias secuencias DIF que, a su vez, se componen de un número entero de bloques DIF de 80 *bytes*, siendo estos las unidades básicas de transmisión DV. Hay cinco tipos de bloques: cabecera DIF, secuencia DIF, subcódigo, vídeo auxiliar, audio y vídeo. Estos bloques, DV-DIF, pueden ser almacenados en formato RAW, cuyos archivos tienen extensiones .dv y .dif, y son utilizados en producciones profesionales de televisión. Estos bloques también pueden ser embebidos en contenedores AVI de Microsoft, QuickTime de Apple o MXF —utilizado, actualmente, en soportes como P2 de Panasonic y en la serie de productos XDCAM/XDCAM EX de Sony. **[1]**

2.2 GRABACIÓN HELICOIDAL

La grabación helicoidal es una técnica de grabación electromagnética que usa como soporte de almacenamiento la cinta magnética, siendo la tecnología de grabación en el sistema DV. La técnica consiste en inclinar ligeramente el tambor que contiene los cabezales respecto al eje longitudinal a la dirección de la cinta. El resultado es que las pistas magnéticas no se trazan de

manera longitudinal y continua a la cinta, sino de forma oblicua y discontinua, consiguiendo un mejor aprovechamiento de la cinta. En el sistema DV cada cuadro está compuesto por 12 pistas oblicuas y consecutivas en PAL, y 10 en el caso del sistema NTSC. Para conseguir una velocidad de grabación adecuada, el tambor gira respecto a su eje longitudinal en sentido opuesto al desplazamiento de los fotogramas, quedando la cinta enrollada en forma de hélice.

Los pares de cabezales se disponen en posiciones opuestas en la superficie del cilindro del tambor, siendo estos los elementos electromagnéticos que graban estas pistas y que están dispuestos, simétricamente, en posiciones opuestas unos de otros. **[2]**

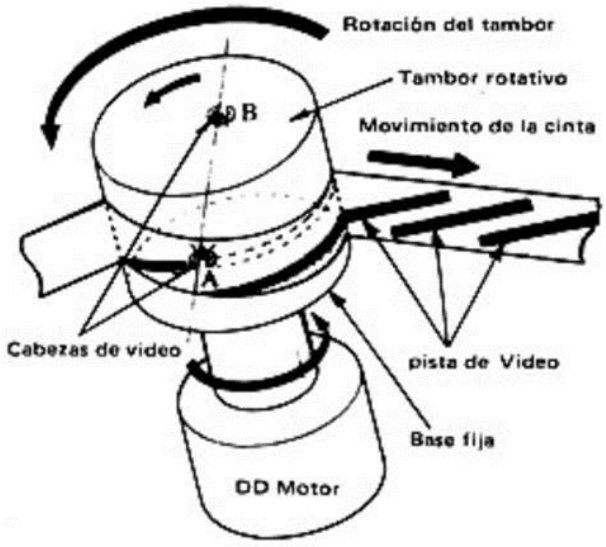


Ilustración 3: grabación helicoidal. Fuente: www.fotolog.com

2.3 FORMATOS DV

El estándar DV es aplicado en diferentes variantes desarrolladas, principalmente, bajo las marcas comerciales de Sony y Panasonic, que, aun manteniendo el mismo esquema de compresión, mejoran en ciertos aspectos el estándar original.

DVC:

Es el formato que más se extendió en cámaras de vídeo doméstico en la década de los 90. Se comercializa en dos tamaños de cinta: DV y Mini-DV. El soporte Mini-DV, debido al pequeño tamaño de las cintas, permitió la fabricación de videocámaras digitales más compactas y

ligeras que grababan vídeo digital en definición estándar. Actualmente, es un formato que tiende al desuso por las razones que se exponen en el siguiente apartado. **[3]**

A continuación se detallan las características principales:

Tabla 1. DV. Características técnicas

Fabricantes	Consorcio de 60 fabricantes	
Uso	Vídeo de consumo	
Resolución/submuestreo de crominancia	SD: 720x480/4:1:1 (NTSC) SD: 720x576/4:2:0 (PAL)	
Compresión	5:1	
Bitrate	25 Mb/s	
Profundidad de color	8 bits	
Soporte	Cinta ¼" 6,33mm	
Ancho de pistas	10μm – SP y 6,7μm - LP	
Canales de audio	2 canales PCM	4 canales PCM
Muestreo de audio	48 KHz/16 bits	32 KHz/12 bits
Velocidad de cinta	18,812 mm/s	
Compatibilidad	Puede grabar DV y puede ser reproducida en DV, DVCAM y DVCPRO	
Emulsión magnética	ME. Metal evaporado	
Duración máxima de grabación	Mini DV 80 a 120 minutos Std. DV 270 minutos	

DVCAM:

Fue la respuesta semiprofesional de Sony que, por aquel entonces, a mediados de los 90, dominaba el mercado de vídeo digital profesional con el formato Betacam Digital; con este nuevo formato, competía con Panasonic como formato para equipos móviles de noticias ENG en el mercado de la teledifusión.

Cuenta con las mismas características que el sistema DV excepto la velocidad de cinta, que es un 50% mayor, y el ancho de pista, que es también mayor. Permite una sincronización de

audio más precisa, mayor robustez y fiabilidad. Al tener un mayor recubrimiento magnético y capa antifricción reduce el *drop-out* un 50% y mejora 5 veces la tasa de error. **[3]**

Tabla 2. DVCAM. Características técnicas

Fabricantes	Sony	
Uso	Profesional/Industrial/ENG	
Resolución/submuestreo de crominancia	SD: 720x480/4:1:1 (NTSC) SD: 720x576/4:2:0 (PAL)	
Compresión	5:1	
Bitrate	25 Mb/s	
Profundidad de color	8 bits	
Soporte	Cinta ¼" 6,33mm	*Professional Disc* y disco duro
Ancho de pistas	15µm	
Canales de audio	2 canales PCM	4 canales PCM
Muestreo de audio	48 KHz/16 bits	32 KHz/12 bits
Velocidad de cinta	28,812 mm/s	
Compatibilidad	Puede grabar DVCAM, reproduce DV y puede ser reproducida en algunas DVCPRO y algunas DV	
Emulsión magnética	ME. Metal evaporado	
Duración máxima de grabación	40-180 minutos	

DVCPRO:

Es la respuesta semiprofesional de Panasonic del estándar DV. Permite una mayor calidad que el estándar DV al aumentar la velocidad y el ancho de la cinta. Fue desarrollado como formato para equipos ENG, permitiendo una calidad considerable con un peso reducido y de fácil portabilidad. Además de la versión estándar, en 1997 se desarrolló el formato DVCPRO 50, el cual, duplicando la velocidad de la cinta, aumentaba el régimen binario y reducía la duración de esta. En 1998, Panasonic desarrolló el formato DVCPRO HD para la televisión de alta definición de EE UU, siendo también utilizado, posteriormente, por medios como la BBC para series en HD **[3].**

Tabla 3. DVCPRO 25. Características técnicas

Fabricantes	Panasonic, Philips, Ikegani, Hitachi	
Uso	Profesional/Industrial/ENG/*Broadcast*	
Resolución/submuestreo de crominancia	SD: 720x480/4:1:1 (NTSC) SD: 720x576/4:1:1 (PAL)	
Compresión	5:1	
Bitrate	25 Mb/s	
Profundidad de color	8 bits	
Soporte	Cinta ¼" 6,33mm	Tarjetas P2
Ancho de pistas	18µm	
Canales de audio	2 canales PCM	
Muestreo de audio	48 KHz/16 bits	
Velocidad de cinta	33,812 mm/s	
Compatibilidad	Puede grabar DVCPRO, reproduce DV	
Emulsión magnética	MP. Partículas metálicas	
Duración máxima de grabación	63-184 minutos	

Tabla 4. DVCPRO 50. Características técnicas

Fabricantes	Panasonic, Philips, Ikegani, Hitachi	
Uso	Profesional/Industrial/ENG/*Broadcast*	
Resolución/submuestreo de crominancia	SD: 720x480/4:2:2 (NTSC) SD: 720x576/4:2:2 (PAL)	
Compresión	3.3:1	
Bitrate	25 Mb/s	
Profundidad de color	8 bits	
Soporte	Cinta ¼" 6.33mm	Tarjetas P2
Ancho de pistas		
Canales de audio	4 canales PCM	
Muestreo de audio	48 KHz/16 bits	
Velocidad de cinta	67,7 mm/s	

Compatibilidad	Puede grabar DVCPRO 50, reproduce DV y DVCPRO 25
Emulsión magnética	MP. Partículas metálicas
Duración máxima de grabación	92 minutos

Tabla 5. DVCPRO HD. Características técnicas

Fabricantes	Panasonic, Philips, Ikegani, Hitachi	
Uso	Profesional/Industrial/ENG/*Broadcast*	
Resolución/submuestreo de crominancia	HD: 1080i y 720p/4:2:0	
Compresión	6,7:1	
Bitrate	25 Mb/s	
Profundidad de color	8 bits	
Soporte	Cinta ¼" 6.33mm	Tarjetas P2
Ancho de pistas	18µm	
Canales de audio	2 canales PCM	
Muestreo de audio	48 KHz/16 bits	
Velocidad de cinta	135,280 mm/s	
Compatibilidad	Puede grabar DVCPRO HD, reproduce DV y DVCPRO	
Emulsión magnética	MP. Partículas metálicas	
Duración máxima de grabación	46 minutos	

2.4 SITUACIÓN ACTUAL

Como se ha comentado en el primer apartado, el soporte para el que en un principio se desarrolló el estándar fue la cinta magnética que, con el paso del tiempo, ha ido perdiendo peso comercial, principalmente con la aparición de nuevos soportes como son los discos duros, las memorias en estado sólido y los soportes ópticos. Los principales inconvenientes de las cintas magnéticas son: **[4]**

- Es un sistema lineal, por lo que para acceder a un punto determinado tiene que recorrer la sección de la cinta desde el punto de partida hasta el punto deseado
- Para transferirse a un PC se debe realizar una captura
- Tanto los campos electromagnéticos como los agentes atmosféricos pueden alterar el campo magnético de la cinta y borrar parte del contenido
- Al reproducir el contenido de la cinta hay un contacto físico entre la cinta y el cabezal, lo que produce un desgaste del soporte
- *Drop-out* durante la grabación debido a desperfectos
- Número limitado de generaciones

Las alternativas al soporte convencional que, actualmente, lo están sustituyendo comercialmente son:

- Memorias de estado sólido: permiten la grabación no lineal de vídeo con altas tasas de escritura, con un número muy alto de generaciones y son muy resistentes a los agentes externos. Los principales inconvenientes son su baja capacidad de almacenamiento y su alto coste
- Discos duros: permiten también la grabación no lineal con altas tasas de grabación y grandes capacidades de almacenamiento a un bajo coste. Los inconvenientes principales son que contienen elementos mecánicos susceptibles de generar ruido y averías que, al ir incorporados en la cámara, no son de fácil sustitución y reparación
- Soportes ópticos: el principal problema es que no permiten un volcado al PC como es el caso de los soportes anteriores y necesitan ser manipulados adecuadamente por su tendencia al deterioro si se hace un mal uso de ellos. Tienen un número alto de generaciones y no sufren desgaste al ser reproducidos, ya que no existe fricción entre el disco y el lector

Una vez vistas las alternativas que actualmente están sustituyendo a los soportes convencionales, vamos a ver cuáles son las actuales tecnologías que usan el formato DV.

XDCAM: fue desarrollado por Sony y adoptado posteriormente por JVC como serie comercial de videocámaras de grabación no lineal. Estas videocámaras utilizan el formato DVCAM (MXF, DV-AVI) para la grabación de vídeo en resolución estándar. Estás videocámaras utilizan como medio de adquisición y almacenamiento diferentes soportes, como son el *Professional Disc*

(*Blu-ray*) o memorias de estado sólido como SxS o la tradicional *Memory Stick* de Sony adecuadamente adaptada.

P2: fue diseñada en 2004 por Panasonic para su línea de productos profesionales como soporte de memoria en estado sólido y adaptada para los equipos ENG. Los formatos soportados por esta tecnología son DVCPRO, DVCPRO 50 y DVCPRO HD. El inconveniente de la capacidad descrito anteriormente en memorias de estado sólido ha hecho que Panasonic haya diseñado sus videocámaras con varias ranuras que evitan cortes en la grabación. Los archivos, al igual que en la tecnología de su competidor XDCAM, son embebidos en contenedores MXF.

Analizando lo anteriormente expuesto, el futuro próximo del formato está garantizado porque Panasonic ha apostado por él para sus tecnologías de vídeo digital HD, resolución cada vez más extendida en producciones de teledifusión y cine digital. En el caso de Sony y JVC, al apostar por el formato de compresión MPEG para sus aplicaciones HD, no se augura una apuesta firme por este formato de cara al futuro.

Professional Disc (Soporte óptico) **P2 (Memoria en estado sólido)**

Ilustración 4: soportes actuales de formatos DV

3. ANÁLISIS DE LA VERSIÓN PREVIA DE SAVID

La versión desde la que parte este proyecto de *software* permite seleccionar, a la entrada del sistema de procesado, un archivo de vídeo comprimido en formato DV que contiene dos fotogramas codificados. El código resultante tras la demodulación y decodificación de los dos cuadros es archivado en ficheros .dat. El procesado de los dos cuadros se realiza de una forma independiente y en paralelo, lo que permite comparar ambos resultados y poder analizar las diferencias. El código binario resultante de la decodificación es almacenado en diferentes archivos en función de su contenido y se distinguen los siguientes tipos:

- Audio
- Subcode
- Vídeo
- Vídeo Auxiliar
- Sector de audio
- Sector EditGap 1
- Sector EditGap 2
- Sector EditGap 3
- Sector Subcode

Estos archivos se guardan en una carpeta llamada “PasosIntermedios”, al mismo nivel de la ruta raíz donde se encuentra el ejecutable.

El proyecto MVS está estructurado en las siguientes clases:

- SAVIDLIO
- SAVIDLIODoc (clase documento)
- SAVIDLIOView (clase vista)
- MainFrm (principal)

- SAVIDLIO.res (recursos)
- StdAfx (clase heredada del *framework*)

Los archivos que contienen el código de análisis son:

- Decodificacion2
- SectoresSAVID
- Cdrs
- Sincros
- SAVIDTools

Los archivos donde se declaran e inicializan las variables:

- InicTablas
- DeclareGlobalsSAVIDLIO

Sincros: contiene las funciones específicas para el análisis de sincros, pudiéndose realizar de manera independiente a la decodificación completa.

Actualmente solo se ejecuta la función “AnalizarSincros”, aunque dispone de código para ejecutar las funciones “AnalizarSincrosErroneos” y “CorregirSincrosErroneos” cuando se estudie su utilidad.

Decodificacion2: contiene las funciones más generales del proceso de decodificación completa:

- RealFrameDVC (llamada desde SAVIDLIOView tras el cuadro de diálogo de abrir archivo dvc)
- Real12Traks (llamada desde SAVIDLIOView tras pulsar el botón “Decodificar”)
- Función_principal (llamadas desde SAVIDLIOVIEW tras llamar a Read12Tracks)

Todas estas funciones y métodos, a su vez, llaman a funciones más específicas, cuyo código se encuentra en los archivos “SectoresSAVID”, “SAVIDTools” y “cdrs”.

En esta versión, la presentación mediante IU es deficiente, pues el código existente no se encuentra dentro del método SAVIDLIOView::OnDraw, que Windows se encarga de ejecutar y refrescar automáticamente.

Además, no genera las imágenes de los dos cuadros, ni el análisis del segundo cuadro como en un principio realizaba ya la aplicación.

Se ha observado otra deficiencia en su implementación al realizar un segundo análisis: los bits decodificados en el segundo análisis se guardan al final del resultado del primer análisis, sin sobrescribir el contenido del primero.

3.1IDENTIFICACIÓN DE ASPECTOS CRÍTICOS

En este subapartado se pretenden identificar aquellos puntos que, a priori, se consideran críticos y que, por los riesgos que conllevan, deben ser tratados de un modo especial antes de tomar una determinación. Un tratamiento adecuado de estos aspectos repercute en una gestión del tiempo más eficiente, incidiendo más en el diseño técnico de dichos puntos y ayudándonos a tomar decisiones justificadas que no nos hagan incurrir en esfuerzos innecesarios que nos obliguen a dar pasos atrás una vez el *software* haya llegado a cierto grado madurez.

Antes de desglosarlos, a continuación se enumeran los aspectos críticos identificados:

- Prolongación del desarrollo del proyecto en un largo periodo de tiempo
- Tecnología MFC como requisito principal
- Archivo de resultados de la decodificación en formato de texto binario .dat

Sobre el primer punto se pueden identificar dos aspectos a tener en cuenta y que son comunes en todos los proyectos cuyo desarrollo, por diferentes razones, se prolonga en el tiempo.

Por un lado, la incompatibilidad manifiesta entre *frameworks* de diferentes versiones de IDE, herramienta de desarrollo integrada que no permite la integración y actualización de antiguos desarrollos con las últimas versiones. Las nuevas versiones no solo facilitan el desarrollo con nuevas funcionalidades, sino que aportan herramientas que mejoran la calidad del *software*.

Por otro lado, el hecho de que se prolongue en el tiempo va asociado, implícitamente, a que un gran número de programadores con diferentes criterios participen en su desarrollo, aportando sus buenas y malas prácticas. Por tanto, la falta de homogeneidad en la versión original dificulta la realización de evolutivos.

Esto nos lleva a tener que decidir entre modular el *software* en diferentes partes, desarrollando las últimas funcionalidades con las últimas versiones del IDE, o renunciar a todas las ventajas que nos aportan y seguir desarrollándolo en la versión de IDE obsoleta con la que se inició la herramienta.

Otro aspecto crítico, y uno de los requisitos principales, es que la tecnología de desarrollo sea MFC (*Microsoft Foundation Classes*). Lo que buscamos en este proyecto es informar mediante IU (Interfaz de Usuario) sobre los resultados del análisis de los datos decodificados, permitiendo al usuario la gestión de reportes de estos resultados. Las aplicaciones MFC programadas en C++ se estructuran de forma predeterminada con una arquitectura documento-vista enfocada a visualizar, editar, gestionar impresiones y guardar diferentes vistas de un documento. En sus primeras versiones no permitía el diseño de interfaces de usuario con asistente GUI, lo que conlleva pérdidas de tiempo innecesarias y diseños desafortunados.

Otras tecnologías como .Net, cuyo CLR, *Common Leguage Realtime,* integra varios lenguajes —entre ellos C++— permiten el desarrollo rápido de aplicaciones de cliente pesado. La IDE incorpora un asistente GUI integrado, tecnologías de hojas de estilo dinámicas como Telerik, fácil gestión de reportes con diferentes formatos como Crystal Report y la fácil estructuración en arquitectura de tres capas, que sería adecuada para el fin del proyecto.

Por último, la versión desde la que partimos en el presente proyecto crea y guarda archivos de texto plano binario como hemos descrito anteriormente, siendo estos archivos los que la nueva versión abrirá y de donde extraerá los datos a analizar. El inconveniente que nos encontramos es que este formato no es el que presta mayor flexibilidad de manipulación, por lo que podría sumarse como inconveniente.

3.2 CONCLUSIONES

Tras identificar y describir los puntos críticos tenemos información suficiente para justificar la estrategia a seguir, la cual se describe a continuación.

La última versión de la que parte el presente desarrollo fue programada con la versión de Microsoft Visual C++ 6.0, presentada en 1992, y que daba soporte integrado de C++ y MFC

6.0. La falta de compatibilidad entre desarrollos de versiones obsoletas y las últimas versiones de IDE nos hace buscar una solución para aprovechar las ventajas del IDE Microsoft Visual Studio 2010 y la funcionalidad de la anterior versión, por lo que la solución pasa por independizar el nuevo desarrollo de la versión de la que partimos.

Las ventajas principales de usar MVS 2010 son:

- Herramientas de trazabilidad que permiten el análisis del código línea a línea y la asignación de variables, como es su depurador *debuger* más evolucionado
- Eliminación de *warning* del código por medio de herramientas integradas de calidad de código como Style Code
- Diseño más atractivo de interfaces de usuario gracias al uso de un asistente GUI
- Poder desarrollar la nueva aplicación con la versión de MFC 10.0
- Permitir la integración del proyecto MFC con otros *frameworks* como .Net que, puntualmente, podría ser utilizado
- Herramienta de ayuda de autocompletado, Intellisense, más sofisticada

Los objetivos que se persiguen en la integración de la antigua versión son:

- Mantener la funcionalidad del desarrollo anterior
- Tratar de ser lo menos intrusivo posible en la antigua aplicación
- Considerar el desarrollo anterior, desde el nuevo desarrollo, como una caja negra que permita, al llamarla, obtener a su salida los archivos binarios con los datos decodificados

Respecto a que MFC sea la tecnología de implementación como requisito principal nos hace descartar el uso de .Net, que podría haber sido una buena solución. Aun así, el uso de MVS 2010 que, como hemos comentado, integra entre otras herramientas un asistente GUI y la posibilidad de crear un proyecto de tipo “cuadro de diálogo”, nos permite realizar un desarrollo con una arquitectura de tres capas y no la de documento-vista, que es inadecuada para el fin de la aplicación.

La utilización de métodos heredados, que abren los archivos .dat y recuperan los bits en codificación hexadecimal, hace que la manipulación de los ficheros no sea un inconveniente, por lo que no se modificarán los formatos de los ficheros en la aplicación original.

4. ARQUITECTURA E INTEGRACIÓN ENTRE APLICACIONES

La arquitectura por defecto de los proyectos MFC, que se verá en detalle en el apartado que describe esta tecnología, es la de documento-vista. Esta arquitectura, como se explicará, no es la más adecuada para el fin de esta herramienta, por lo que el prescindir de los proyectos tipo SDI y MDI y optar por un proyecto "cuadro de diálogo" para la implementación ha permitido estructurar la aplicación en una arquitectura de tres capas.

Esta arquitectura paradigmática es una de las más utilizadas en este tipo de aplicaciones y, en ella, se diferencian las capas:

- DAL (Capa de acceso a datos): es la capa de nivel inferior, que permite acceder y recuperar los datos, en este caso, de los ficheros binarios y, habitualmente, de las BBDD. Por lo tanto, es la capa que abre los ficheros y almacena el contenido de estos en arreglos sin realizar un procesado de los bits. En la aplicación, esta capa es llamada SAVIDLIO_DAL
- BL (Capa de negocio): es la capa intermedia, donde se procesa la información recuperada por la DAL, por lo tanto, es donde, por medio de estructuras condicionales, se asigna el significado sustraído de los bits codificados en función de las especificaciones de la norma. En la aplicación, esta capa es llamada SAVIDLIO_BL
- IU: la interfaz de usuario es la capa superior y permite al usuario interaccionar con la aplicación, de manera que el sistema le muestra los resultados de sus peticiones. En la aplicación, las IIUU se corresponden con los cuadros de diálogo de cada una de las pestañas

Las dos aplicaciones se dividirán, funcionalmente, en:

- Aplicación antigua (SAVID Procesador): es la encargada de realizar la demodulación, decodificación y archivo en ficheros de los datos de vídeo y audio decodificados

- Aplicación nueva (SAVID Visualizador y reportador de resultados): realiza las funciones de recuperar los archivos, mostrar los resultados del análisis y la gestión de reportes

Descripción:

La solución final tendrá dos módulos correspondientes a los dos desarrollos independientes. Desde la nueva aplicación se puede ejecutar la antigua versión por medio de un evento de botón que llame al ejecutable de SAVID Procesador. La antigua versión ha sido modificada reduciéndose a un *pop-up* que permite buscar la ruta donde se encuentra el archivo de vídeo. Una vez seleccionado, se realiza el proceso de decodificación de los dos cuadros que contiene el fichero de vídeo, guardándose los resultados en archivos diferentes dependiendo del tipo de contenido y en la carpeta anteriormente referenciada llamada "PasosIntermedios". De una forma secuencial, una vez se ha realizado todo el proceso, la aplicación se cierra automáticamente.

Tras ejecutar SAVID Procesador y crear la carpeta con los ficheros que contienen los bits de los sectores de vídeo auxiliar, audio auxiliar y *edit gap,* siendo estos los ficheros que analizaremos, el sistema mostrará la información procesada, una vez el usuario pulse el botón "Analizar" o asigne un cuadro en el radio *button* homónimo en las pestañas "Vídeo Auxiliar" y "Audio Auxiliar" o informados los combos "Pista" y "Nº Edit Gap", y el radio *button* Cuadro, en la pestaña "Edit Gap".

En el siguiente esquema se ilustran las relaciones entre las dos aplicaciones y las llamadas correspondientes.

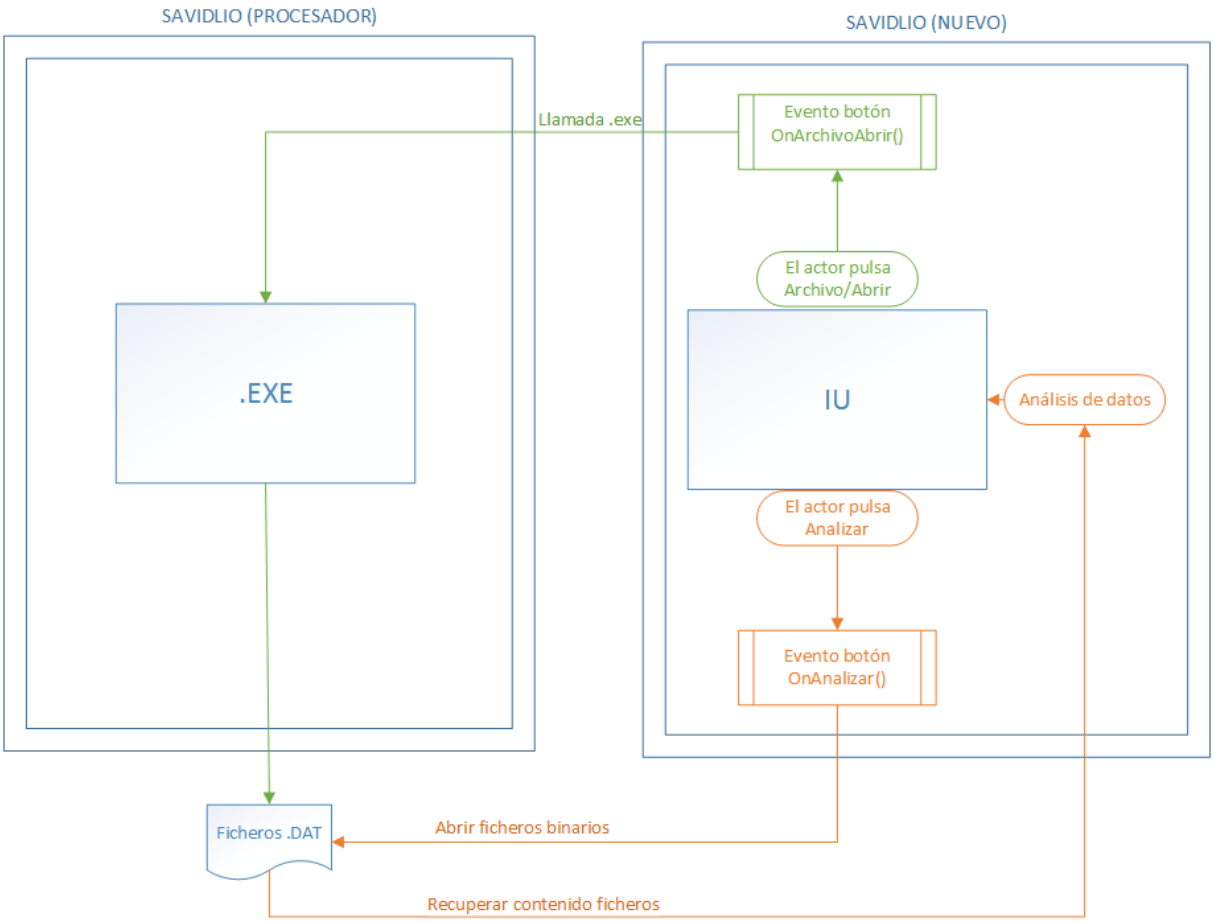


Ilustración 5: esquema de integración con aplicación previa

5. CALIDAD DE *SOFTWARE*

Aunque la calidad del *software* es considerada de vital importancia en el desarrollo de aplicaciones profesionales, en la práctica se obvia en la mayoría de los casos. El hecho de que las buenas prácticas repercutan en el aumento de costes hace que, en muchas ocasiones, las partes implicadas se conformen con que una aplicación cumpla con los requisitos sin importar cómo se llegó a la solución final. Los problemas suelen llegar cuando el desarrollo del *software* va más allá de lo que en un principio se pensó y surgen problemas diversos, como son que el volumen de datos sea mayor del que se consideró en un primer momento, mostrando el sistema ineficiencias al recuperarlos o que al realizar evolutivos, si no han sido correctamente implementados conceptos como la encapsulación, se complique la reutilización y la modificación del código.

En conclusión, lo que en un principio puede considerarse un mal menor puede terminar siendo un despropósito, por lo que, siguiendo con el objetivo del proyecto y sin caer en trivialidades, se tratará de desarrollar sobre la base de unas buenas prácticas que garanticen, en la medida de lo posible, la sostenibilidad y el desarrollo de futuros evolutivos.

5.1 ISO-9126

La calidad, en el ámbito de la ingeniería de *software*, no solo se mide con el hecho de que la aplicación funcione y cumpla con los requisitos del usuario, sino con el cumplimiento de una serie de características contempladas en el estándar ISO-9126, que son las siguientes:

- Funcionalidad:
    - Adecuación
    - Exactitud
    - Interoperabilidad

    - Seguridad de acceso
- Fiabilidad:
    - Madurez
    - Tolerancia a fallos
    - Capacidad de recuperación
- Usabilidad
    - Capacidad para ser entendido
    - Capacidad para ser aprendido
    - Operatividad
    - Capacidad de atracción
- Eficiencia
    - Comportamiento temporal
    - Uso de los recursos del sistema
- Sostenibilidad
    - Capacidad para ser analizado
    - Estabilidad
    - Capacidad para ser probado
- Portabilidad
    - Adaptabilidad
    - Instalabilidad
    - Coexistencia
    - Capacidad para ser reemplazado

La llamada calidad demostrable, que se abarca en el apartado de fase de pruebas, trata de realizar una serie de pruebas programadas que den garantías del cumplimiento de estas características que definen la calidad. **[5]**

5.2 TÉCNICAS QA

A continuación, se describen algunas técnicas implementadas en ingeniería de *software* para garantizar la calidad de herramientas informáticas, cada vez más extendidas en el sector IT, en parte por la importancia que le han dado las metodologías ágiles.

Desarrollo Guiado por Pruebas (TDD): *Test driver development* [6]:

Es una técnica de diseño de *software* que sigue con el siguiente ciclo:

1. Diseño y desarrollo de pruebas unitarias
2. Implementación del código
3. Refactorización

Lo convencional es, en primer lugar, implementar los requisitos programando el código para realizar las pruebas en segundo lugar; en cambio, con esta técnica primero se diseñan las pruebas unitarias que verifican un requisito para programar el código que haga que la prueba no falle. Es importante, como en cualquier técnica de desarrollo, la refactorización del código y las posteriores pruebas que garanticen que no se ha modificado la funcionalidad. Otra característica de esta técnica es la automatización de las pruebas, independizándolas así de la subjetividad humana del ejecutor, permitiendo también la iteración de las mismas a un bajo coste.

Las ventajas principales de describir primero las pruebas y a partir de estas realizar el desarrollo son, por un lado, que se obliga a realizar de manera indirecta un análisis más exhaustivo de los requisitos y, por otro, que se evita que las pruebas estén condicionadas a la solución implementada por el desarrollador, que puede obviar escenarios no previstos.

Otra característica de esta técnica es la flexibilidad, que le permite ser utilizada con cualquier metodología de desarrollo, como son el método en cascada o el método XP —*eXtreme Programming*—, siendo este último el que la puso de actualidad cuando Kent Beck la introdujo como parte de la metodología. También se consigue que no se escriba código redundante e innecesario, al codificarse solo el código suficiente para que no falle la prueba unitaria.

En la práctica, no se pasa a una nueva funcionalidad en caso de que el *software* no haya pasado un test, ni se considera implementar una funcionalidad en la que no se pueda definir una prueba que la verifique.

Como podemos ver, esta técnica cumple con las metodologías ágiles de desarrollo incremental. En el siguiente esquema se representa el ciclo que se lleva a cabo con ella:

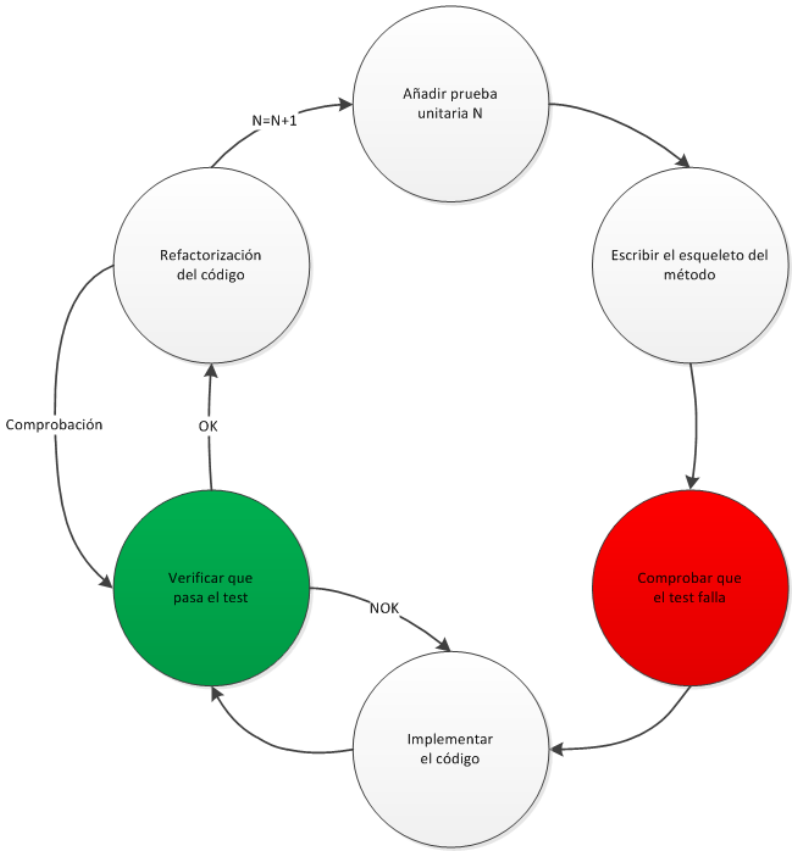


Ilustración 6: técnica TDD

Desarrollo Guiado por Pruebas de Aceptación (ATDD) [7]:

La técnica TDD es un desarrollo guiado desde el ámbito técnico del programador, por lo que, aun verificando que la solución funciona correctamente, nunca podría validar que el desarrollo cumple con lo deseado por el usuario. La razón de esta afirmación es que, aunque se haya partido de casos de uso que describen la funcionalidad, definidos por el cliente y el analista, en estos casi siempre se cae en la malinterpretación y en el error por una falta de detalle que hace que no cumpla con los criterios de aceptación del cliente. La técnica ATDD, en cambio, es un desarrollo guiado directamente por el lado del cliente con la ayuda del desarrollador, pudiendo este realizar un seguimiento del desarrollo comprobando si se está cumpliendo con los test de validación que definen la funcionalidad.

En las metodologías convencionales, el analista de negocio escribe todos los requisitos en archivos de texto, mientras que con esta técnica, siguiendo la línea de las metodologías ágiles, dichos archivos se sustituyen por documentos ejecutables escritos con lenguajes definidos que permiten ser procesados por herramientas xUnit.

Los casos de uso tradicionales son sustituidos por las llamadas historias de usuario, con la diferencia de que estas no pretenden hacer una descripción precisa de la funcionalidad que sea susceptible de ambigüedades y malinterpretaciones, sino que definen el requisito por medio de ejemplos. Estas historias deben ser nombradas con un lenguaje de fácil comprensión, siempre alejado de lenguajes técnicos de desarrollo, que resuman en una sola frase la funcionalidad requerida.

De cada uno de los enunciados que sintetizan el requisito en una frase y de las interacciones entre desarrolladores y clientes surgen una serie interrogantes sobre los mismos, y las respuestas a estas preguntas son los ejemplos que definen los test de aceptación.

La labor de los desarrolladores y del ejecutor de las pruebas es traducir estos ejemplos (test de aceptación) en lenguaje formal al lenguaje con el que trabaje la herramienta xUnit y que, como hemos indicado anteriormente, permita automatizar las pruebas.

Esta técnica, en caso de ser implementada, se ejecuta junto a la técnica TDD, por lo que para que pase un test de validación antes tiene que pasar una serie de pruebas unitarias. La gran virtud de esto es que, por un lado, solo se va a implementar el código necesario para que cumpla con el test unitario, sin caer en el error de escribir "código muerto" y, por otro, al comprobar si cumple con los requisitos de validación cada vez que termina un ciclo TDD, no se cae en la mala práctica de añadir funciones que nunca serán utilizadas.

En definitiva, es una técnica que se centra en que el cliente, por medio de un test, exprese "qué" es lo que quiere y no "cómo" lo quiere, labor que siempre tendría que caer del lado del desarrollador y que, con otros métodos, no siempre es así. Es una manera fácil de aportar valor por parte del desarrollador.

Resumiendo, los pasos a seguir con esta técnica son:

- Se define una "historia de usuario"
- Se hacen preguntas sobre la historia y la respuesta del cliente/usuario es la que define el test de validación
- Se traduce al lenguaje de la herramienta para automatizar pruebas
- Se itera el ciclo TDD tantas veces como sea necesario hasta que cumpla la validación

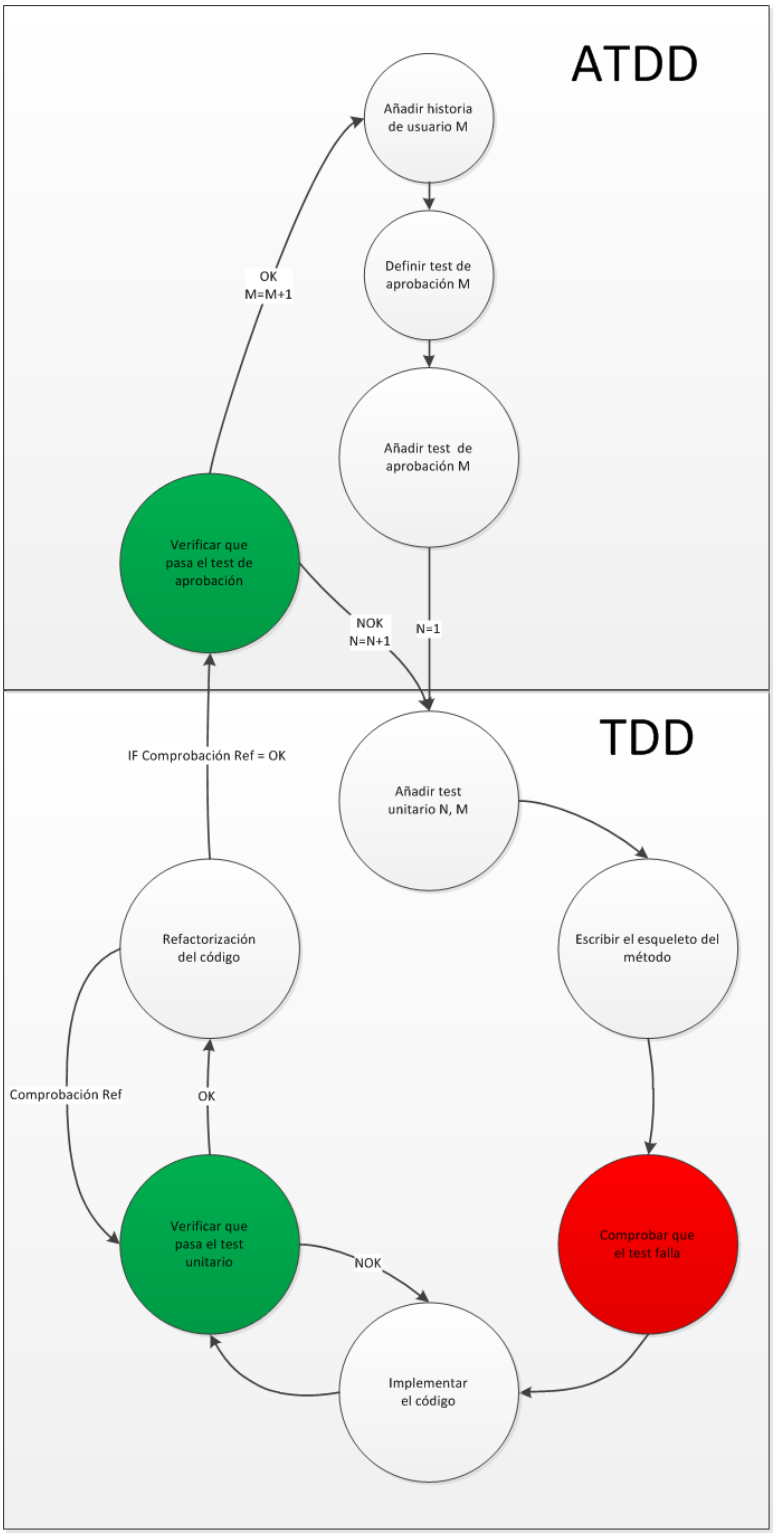


Ilustración 7: técnicas ATDD y TDD

Desarrollo Guiado por Comportamiento (BDD) [8]:

Al igual que la técnica ATDD, el fin de esta es cubrir con la validación por parte del usuario de la que adolece la técnica TDD. En la técnica BDD, desarrollada por Dan North, se redactan una

serie de “historias” en el lenguaje natural Gherkin, que permite una comunicación entre el desarrollador y el encargado de especificar los requisitos, facilitando la comunicación y el entendimiento entre ambos. Estas historias que describen las funcionalidades son las que dirigen el desarrollo, estando escritas en un lenguaje definido que permite ser procesada por herramientas de *software*.

Las historias describen las características que se van a desarrollar por áreas de dominio. En el lenguaje definido se escribiría como “*As a* [rol] *I want* [característica] *so that* [beneficio]”. Por ejemplo: “Como un [Administrador] quiero [Escribir un correo] para [Enviarlo] “

Posteriormente, estas características se describen con los diferentes escenarios posibles y los pasos que se darían en ese caso. Estos pasos son definidos como:

- *Given* … (precondiciones)
- *When* … (acciones)
- *Then* … (resultados)

También es complementaria a la técnica TDD y cumple con un esquema similar al de la “Ilustración 7: técnicas ATDD y TDD”.

Algunas herramientas son Cucumber para PHP o para C++, que es el lenguaje utilizado en el proyecto, Igloo BDD, Seleniun o MSpec.

Como se puede observar, las similitudes entre ATDD y BDD son manifiestas y las diferencias entre ambas son más de carácter filosófico, ya que su finalidad es permitir validar el *software* de una forma automática por medio de herramientas. En ambas se fomenta la participación activa de los clientes en el desarrollo, tanto en su definición como en el seguimiento.

Integración Continua [9]:

Esta técnica trata de asegurarse de que el nuevo código no afecta al código programado previamente, por lo que, con la ayuda de *software* que automatiza estas pruebas, se implementan y compilan los test con cierta frecuencia de toda la solución guardada en el CVS (Sistema de control de versiones), reportando los resultados de las pruebas.

Así, se obtienen informes continuos de la situación real del desarrollo verificado, por lo que en caso de la aparición de errores se pueden identificar y depurar antes de que finalice el

desarrollo. Además, se puede tener una versión estable en cualquier punto del *sprint* en caso de metodologías ágiles.

En resumen, la integración continua se basa en obtener la última versión del CVS, compilarla, ejecutar las pruebas, en el almacenamiento de binarios y en la generación de informes basándose en estos.

Actualmente, hay plataformas que ofrecen integración continua en la nube como, por ejemplo, Bamboo de Atlassain o CloudForge de CollabNet.

Programación en pareja [10]:

La programación en pareja o programación a pares es una técnica que se basa en que dos desarrolladores trabajen conjuntamente en un mismo terminal. Mientras uno conduce el plan de pruebas, el otro navega, cambiándose los roles cuando se haya finalizado un bloque.

El conductor es el encargado de escribir el código que funcione y compile mientras que el navegante piensa cómo se va a integrar este código con el resto de la solución, cómo se va a refactorizar el código y qué pruebas se realizarán para verificar su correcto funcionamiento.

Este trabajo colaborativo hace que los desarrolladores conozcan las técnicas de desarrollo del otro, permitiendo la identificación temprana de malas prácticas y posibles errores. Además, al ser roles rotativos, el trabajo se hace menos monótono y el realizarlo en pareja hace que no influyan las interrupciones externas a la productividad de ambos. Como beneficio añadido, en caso de bloqueos es más fácil salir de ellos, al ser dos personas las que piensan en la solución al problema.

Revisiones de código [11]:

Es una alternativa al trabajo a pares en la que cada desarrollador trabaja independientemente en su terminal y, cada cierto tiempo, se realiza una revisión del código del compañero. Esto facilita la detección temprana de errores y, al discutir abiertamente posibles alternativas a la solución implementada, hace que esta pueda ser más eficiente. Con esta técnica se consigue que mejore la calidad del código, permitiendo que las interacciones continuas entre desarrolladores hagan que los equipos ganen en valor al compartir conocimientos de una

forma activa y continua. Esta técnica, al igual que la de desarrollo a pares, es utilizada en metodologías ágiles, aunque puede ser incorporada a otras metodologías.

Herramientas de análisis estático [12]:

Son aquellas que se realizan sobre el propio código fuente sin necesidad de ejecutar el *software*, llevándose a cabo de una forma automatizada por medio de herramientas. Han ido ganando popularidad con los años al tener herramientas más eficientes que, sin sustituir al cien por cien la revisión de los propios desarrolladores, sí que suponen una gran ayuda a bajo coste.

Refactorización [13]:

Esta técnica se basa en la restructuración del código con el fin de eliminar redundancias, mejorar su claridad y su uniformidad y replantearse algoritmos para poder llegar a soluciones más eficientes. En definitiva, sirve para realizar una "limpieza del código".

Por esta razón, no realiza ninguna modificación de la funcionalidad previa y no identifica ni corrige errores. Es una técnica que forma parte de otras, como es el caso de la técnica TDD que se ha descrito anteriormente.

Notación [14]:

El estilo de programación no tiene otro fin que hacer más legible el código, permitiendo así identificar rápidamente constantes, variables, clases, componentes, métodos, etc. Esta homogeneidad en el código facilita su mantenimiento y la implementación de evolutivos. Se han especificado una serie de directrices englobadas en los tipos:

- Notación C: este estilo está actualmente en desuso y la única directriz que especifica es la de sustituir los espacios por el guion bajo con todas las letras en minúscula
- Notación posicional: en este estilo se nombra con una serie de identificadores que contienen en su conjunto "aplicación + lenguaje + subproceso + secuencia"; actualmente está en desuso, al igual que la notación C
- Notación CamelCase: es actualmente la más extendida de todas, con las variantes:
    - UpperCamelCase: la separación entre palabras se elimina y se escribe con mayúscula la primera letra de cada una de ellas

    - LowerCamelCase: la separación entre palabras se elimina y se escribe con mayúscula la primera letra de cada una de ellas, menos la primera que también se escribirá en minúscula
- Notación húngara: es una variante de la notación Camel, que añade al comienzo del identificador unas letras en minúsculas que dan cierta información sobre el componente que identifica

5.3 TÉCNICAS IMPLEMENTADAS EN EL PROYECTO

Tras describir algunas de las técnicas que se llevan a cabo en ingeniería de *software* con el fin de dar valor de calidad a los proyectos de desarrollo, podemos, justificadamente, indicar qué técnica vamos a aplicar en el presente proyecto.

La técnica que se ha elegido es la TDD que, como ya se ha descrito, se basa en guiar el desarrollo sobre la base de pruebas unitarias. Este desarrollo incremental tiene dos debilidades manifiestas: la técnica TDD se basa en pruebas unitarias, por lo que es necesario complementarla con pruebas integradas y de sistema que testean el *software* en su conjunto, las cuales se detallan en el apartado correspondiente a la "Fase de pruebas". Además, que el desarrollo sea incremental tiene el gran inconveniente de que la arquitectura no está predefinida, por lo que se tratará de hacer un diseño previo y, sobre la base del mismo, llegar a un compromiso con estas técnicas.

Por un lado, las técnicas de programación a pares y las revisiones de código, al ser realizarse por parejas, no pueden ejecutarse en este proyecto. Por otro lado, la integración continua, al necesitar una estructura compleja, ha sido descartada, y la posibilidad de uso de la técnica de análisis estático también se descartó al considerar que no añade valor al proyecto.

Cabe destacar también que las técnicas BDD y ATDD no se van a aplicar, debido a que las pruebas de validación del usuario final, como se indicó en los objetivos del proyecto, no se realizarán hasta su finalización, por no tener el material necesario para llevarlas a cabo.

Respecto al estilo de programación elegido, se ha escogido la notación CamelCase para nombrar los identificadores. También se implementará la notación para constantes usada en los lenguajes C++, con todas las letras en mayúscula y sustituyendo los espacios entre palabras por el guion bajo.

6. ANÁLISIS DE DATOS DECODIFICADOS

Tras la demodulación y decodificación del archivo de vídeo, como se describió en apartados precedentes, los bits resultantes se almacenan en ficheros binarios, separándolos en función del origen de la información que contienen. En esta sección, se analizará la información de cada fichero, interpretándola con la ayuda del estándar para grabación de vídeo digital DV SMPTE 306M (Formato-D7), por lo que este apartado será la piedra angular del que partirá el análisis funcional y del que dependerá el cumplimiento de los objetivos finales del proyecto.

Se profundizará en el análisis de los sectores que contengan información de relevancia y que permitan identificar si se ha realizado una edición sobre el contenido original de la cinta.

Los sectores más importantes son los siguientes:

- Cabecera y secciones auxiliares del sector de vídeo
    - Señal de tipo de vídeo
    - Sistema de gestión de generación de copias, *Copy Generation Management System* (CGMS)
    - Modo de visualización
    - Tipo de etiquetas, *flags*, cuadro/campo
- Cabecera y secciones auxiliares del sector de audio
    - Frecuencia de muestreo
    - Cuantificación
    - Modo de audio
    - Número de canales de audio
    - CGMS
    - *Flag* de énfasis
    - Puntos de edición insertados

- Subcódigo
    - Contiene un ID
    - Contiene datos (principalmente TC y grupos binarios)
- Bits de relleno, *edit gap*
    - Los espacios entre los sectores que contienen información no son zonas sin grabar, pero son sectores de datos rechazados en la reproducción
    - Durante la grabación es usada una única secuencia
    - Son parcialmente regrabadas durante la edición, almacenando los errores de sincronización

Ilustración 8: sectores de una pista en formato DV

Las razones por las que el análisis se hace externamente con la aplicación informática son las siguientes:

- Evitar la corrección de errores
- Evitar el rechazo durante la grabación a los datos introducidos que no sean de audio y vídeo
- Estos datos podrían modificarse a lo largo de los procesos de manipulación, como es la edición de la cinta

Toda la información de este apartado ha sido obtenida de la norma referenciada en la bibliografía **[15]**.

6.1 SECTOR DE VÍDEO

El sector de vídeo contiene las secciones Sync, ID, bits de paridad, vídeo auxiliar y datos de vídeo. En particular, la sección que nos interesa analizar es la de Vídeo Auxiliar (VAUX), ya

que es la que contiene los metadatos con la información de las características de la señal de vídeo.

Esta sección se inserta entre los datos de vídeo como se muestra en la siguiente figura resaltado en color rojo.

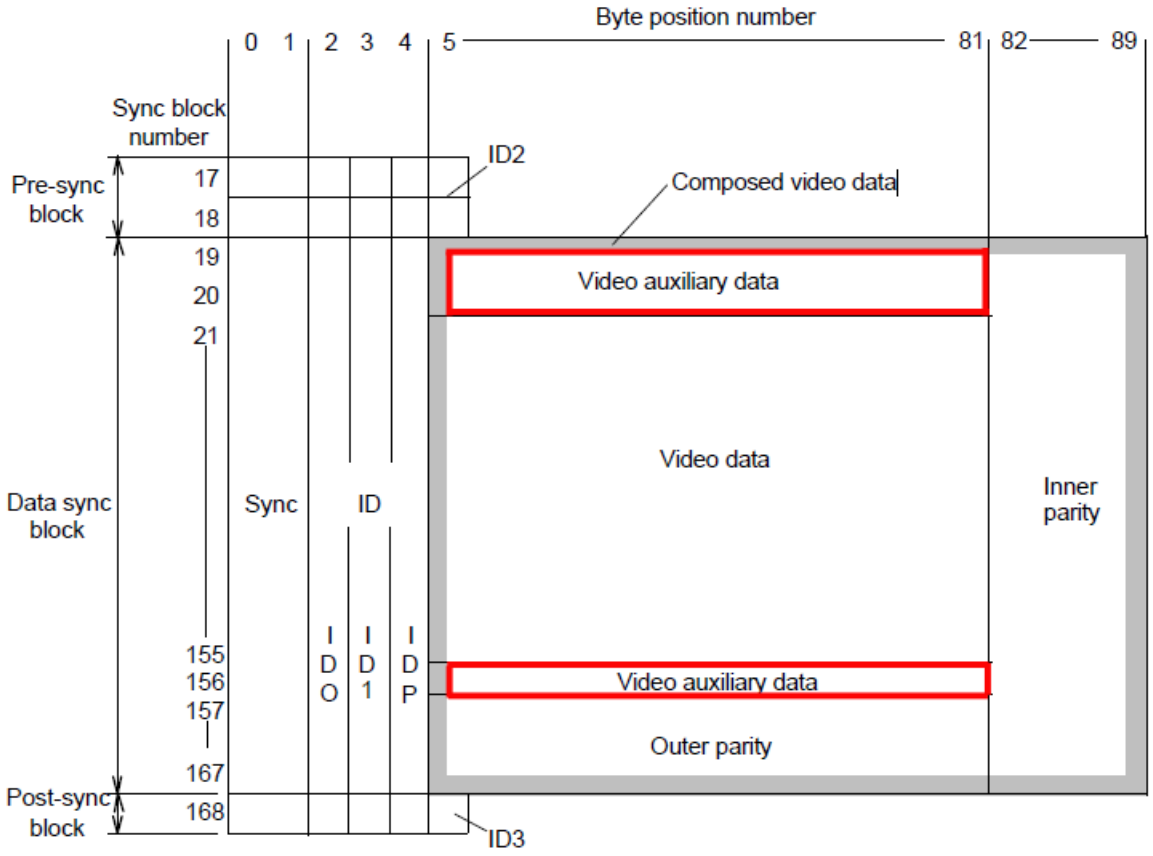


Ilustración 9: sector de vídeo auxiliar. Fuente: SMPTE 306M

El sector de VAUX está estructurado formando paquetes de tamaño fijo de 5 *bytes*, siendo 15 paquetes por cada uno de los bloques de sincronización 19, 20 y 156. Por lo tanto, hay 45 paquetes de vídeo auxiliar por cada una de las 10 o 12 pistas que componen un cuadro. Al estar cada bloque de datos *Sync* compuesto de 77 *bytes* de VAUX y al ser 15 bloques de 5 *bytes* solo 75 *bytes*, a los dos *bytes* restantes se les asigna el valor por defecto de FF hex.

Cada paquete está numerado de forma correlativa del 0 al 44, llamándose “número de paquete de vídeo”. En la siguiente figura se muestra la disposición de cada paquete de vídeo.

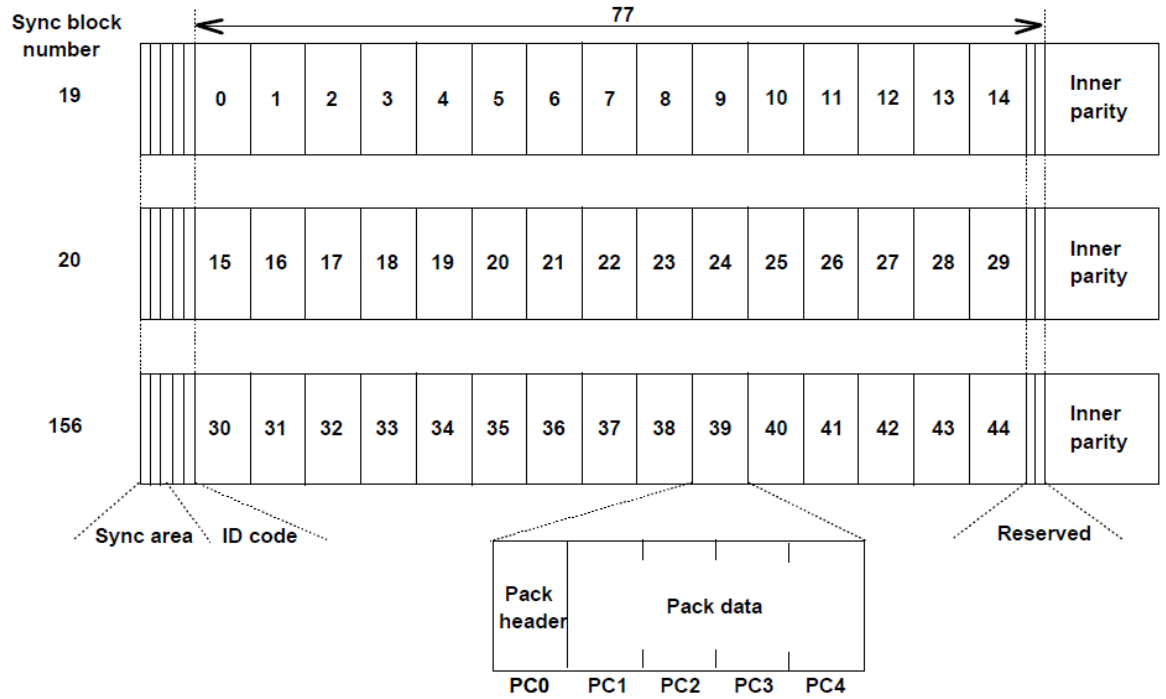


Ilustración 10: paquete de vídeo. Fuente: SMPTE 306M

En la siguiente tabla se muestra la localización de los paquetes VAUX Source y VAUX Source Control, que incluyen los metadatos obligatorios para la reproducción de las señales de vídeo y que deben ser grabados.

Video pack number		VAUX data of a video frame
Even track	Odd track	
39	0	VS
40	1	VSC

NOTE

VS: VAUX source pack (pack header = 60_h)
VSC: VAUX source control pack (pack header = 61_h)

Even track: Track number 0, 2, 4, 6, 8 for 525/60 system
Track number 0, 2, 4, 6, 8, 10 for 625/50 system

Odd track: Track number 1, 3, 5, 7, 9 for 525/60 system
Track number 1, 3, 5, 7, 9, 11 for 625/50 system

Ilustración 11: posición VS y VSC. Fuente: SMPTE 306M

-VAUX Source *pack* (VS)

La siguiente tabla muestra el mapeo del paquete VAUX Source:

	MSB							LSB
PC0	0	1	1	0	0	0	0	0
PC1	Res	Res	Res	Res	Res	Res	Res	Res
PC2	B/W	EN	CLF		Res	Res	Res	Res
PC3	Res	Res	50/60	STYPE				
PC4	0	Res	Res	Res	Res	Res	Res	Res

Ilustración 12: contenido Video Source. Fuente: SMPTE 306M

PC0:

Este *byte* es el identificador con el valor 60 hex para el caso del paquete VS.

PC2:

B/W *Black and white flag*.

Informa si la imagen es en color o en blanco y negro.

- 0 = blanco y negro
- 1 = color

EN *Color frames enable flag*.

Indica si el CLF es válido.

- 0 = CLF es válido
- 1 = CLF es inválido

CLF: *Color frames identification code*.

Es utilizado en vídeo compuesto a la hora de identificar cuál es la secuencia de cuadro para que, en el caso de realizar una edición, se mantenga la sincronización de la subportadora de color con los fotogramas de vídeo en el tiempo determinado por la señal de sincronismo de color.

Para el sistema 525/60:

- 00b = color cuadro A
- 01b = color cuadro B
- Otros = reservados

Para el sistema 625/50:

- 00b = primer y segundo campo
- 01b = tercer y cuarto campo
- 10b = quinto y sexto campo
- 11b = séptimo y octavo campo

PC3:

Stype: definen el tipo de señal de vídeo (submuestreo de croma)

- 00000: para 4:1:1 compresión (D-7)
- 00001 – 11111: reservados

50/60: define el sistema de vídeo

- 0: 50
- 1: 60

VAUX Source Control *pack* (VSC)

La siguiente tabla muestra el mapeo del paquete VAUX Source Control (VSC)

	MSB							LSB
PC0	0	1	1	0	0	0	0	1
PC1	CGMS		Res	Res	Res	Res	Res	Res
PC2	Res	Res	0	0	Res	DISP		
PC3	FF	FS	FC	IL	Res	Res	0	0
PC4	Res	Res	Res	Res	Res	Res	Res	Res

Ilustración 13: Contenido Video Source Control. Fuente: SMPTE 306M

PC0:

Este *byte* es el identificador con el valor 61 hex para el caso del paquete VSC.

PC1:

CGMS: sistema de gestión de generación de copias. Nos indica si están o no permitidas las copias y, en algunas variantes de la norma, el número de copias permitidas.

- 00: copia libre
- 01: TBA (*To be announced*, por asignar)
- 10: TBA (*To be announced*, por asignar)
- 11: TBA (*To be announced*, por asignar)

PC2:

DISP: relación aspecto, formato y posición

- 000: 4:3 *full format*. Posición N/A
- 001: 16:9 *letter box*. Posición central
- 010: 16:9 *full format* (*squeeze*). Posición N/A
- 011 -111: reservados

PC3:

FF *frame/field flag*: indica si los dos campos son entregados o un campo se repite durante un periodo de trama

- 0: solo uno de los dos campos es entregado dos veces
- 1: ambos campos son entregados en orden

FS *first/second flag*: indica qué campo debe ser entregado primero durante un periodo de campo.

- 0: campo 2 (superior)
- 1: campo 1 (inferior)

FC *frame change flag*: indica si el cuadro de la secuencia actual se repite basándose en el cuadro inmediatamente anterior

- 0: misma imagen que el cuadro previo
- 1: diferente imagen que el cuadro previo

IL *Interlace flag*: indica si es vídeo entrelazado o vídeo progresivo

- 0: progresivo
- 1: entrelazado

Resto de VAUX

El resto del área de VAUX se compone de 43 paquetes por pista, 430 paquetes por cuadro para el sistema 525/60 y 516 paquetes por cuadro para el sistema 625/50.

El área reservada de VAUX es como sigue:

- Sistema 525/60 (43 x packs + 6 *byte*s 5 *bytes*) x 10 pistas x 30 cuadros = 66330 *bytes*
- Sistema 625/50 (43 x packs + 6 *bytes* 5 *bytes*) x 12 pistas x 25 cuadros = 66330 *bytes*

El área reservada será cubierta por FFh.

ANÁLISIS DE LOS FICHEROS DE VÍDEO:

A continuación se analizan los ficheros generados tras la descodificación que extrae la información del vídeo de prueba.

Los doce ficheros son llamados como vaux_TT_FFF.dat, donde T indica el número de pista y F el campo de vídeo. En estos se guardan los 77 *bytes* de vídeo auxiliar de cada una de los bloques Sync 19, 20 y 156.

Como se ha descrito anteriormente, se analizarán los 45 bloques de vídeo de 5 *bytes* tanto para pistas pares como impares, para comprobar que realmente cumple con las especificaciones de la norma, como se muestra en las figuras correspondientes a las pistas 0 y 1.

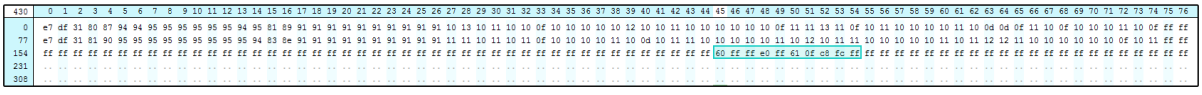


Ilustración 14: posición VS y VSC, pista par. Fuente: SMPTE 306M

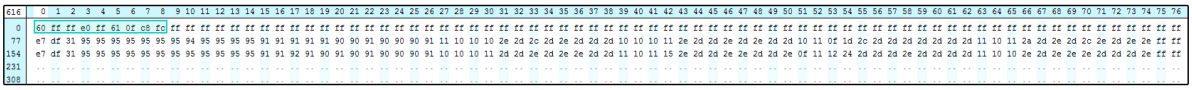


Ilustración 15: posición VS y VSC, pista impar. Fuente: SMPTE 306M

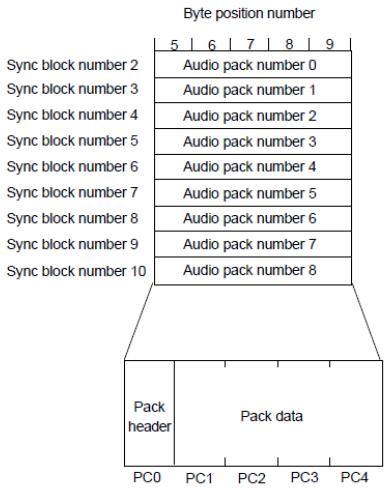
Como podemos comprobar en las pistas pares, en los bloques de vídeo 39 y 40 es donde se encuentran los bloques VS y VSC, correspondiéndose con los *bytes* del 45 al 49 para VS y del 50 al 54 para VSC. En el caso de las pistas impares, en los bloques de vídeo 0 y 1 es donde se encuentran los bloques VS y VSC, que se corresponden con los *bytes* del 0 al 4 para VS y del 5 al 9 para VSC.

6.2 SECTOR DE AUDIO

El sector de audio contiene las secciones Sync, ID, bits de paridad, audio auxiliar y datos de audio. La sección en particular que nos interesa analizar es la de Audio Auxiliar, AAUX, ya que es la que contiene los metadatos con la información de las características de la señal de audio.

Datos auxiliares de audio (AAUX):

Los metadatos que contiene el sector AAUX son añadidos antes que los datos de audio, divididos en 9 bloques de 5 *bytes*. De los 9 bloques solo dos contienen información, el resto están reservados para su uso futuro teniendo todos los bits un valor de FFh. Los dos bloques a analizar son el AS, paquete fuente de AAUX y el ACS, paquete de control fuente de AAUX. En las siguientes figura y tabla se muestra la disposición de estos bloques.



Audio pack number		AAUX data of a video frame
Even track	Odd track	
3	0	AS
4	1	ASC

NOTES

1 AS = AAUX SOURCE pack (pack header = 50_h).

2 ASC = AAUX SOURCE CONTROL pack (pack header = 51_h).

3 Unused AAUX packs shall be reserved.

4 Even track: Track numbers 0, 2, 4, 6, 8 for 525/60 system; Track numbers 0, 2, 4, 6, 8, 10 for 625/50 system.

5 Odd track: Track numbers 1, 3, 5, 7, 9 for 525/60 system; Track numbers 1, 3, 5, 7, 9, 11 for 625/50 system.

Ilustración 16: estructura y posición de VS y VSC. Fuente: SMPTE 306M

Como se observa en la ilustración anterior, para cada pista, el paquete de datos auxiliar de audio corresponde a los bloques de Sync del número 2 al 10 y los *byte*s del 5 al 9 dentro de cada bloque. El *byte* en la posición 5 es el de la cabecera de cada uno de los 9 paquetes, que se numeran del 0 al 8. La posición en la que se encuentran los paquetes AS y ACS depende de la paridad de la pista como se indica en la tabla que acompaña a la ilustración.

Así, para el archivo de prueba que se está analizando, al ser del sistema 625/50 para las pistas numeradas como 0, 2, 4, 6, 8, 10 (*even track*), el paquete 3 se corresponde con el AS y el 4 con el ASC; en el caso de las pistas 1, 3, 5, 7, 9, 11 (*odd track*), el paquete 0 se corresponde con el AS y el 1 con el ASC.

El sector AAUX tiene un área reservada de datos, como se muestra a continuación:

Sistema 525/60: 5 *bytes*/pack X 7 packs/pista X 10 pistas/cuadro X 30 cuadros/s = 10.500 *bytes*/s

Sistema 625/50: 5 *bytes*/pack X 7 packs/pista X 12 pistas/cuadro X 25 cuadros/s = 10.500 *bytes*/s

El área reservada será cubierta por FFh.

AAUX Source *pack* (AS)

En la siguiente ilustración se muestra la estructura de los paquetes AS.

	MSB							LSB
PC0	0	1	0	1	0	0	0	0
PC1	LF	Res	AF SIZE					
PC2	0	CHN		Res	AUDIO MODE			
PC3	Res	Res	50/60	STYPE				
PC4	Res	Res	SMP		QU			

Ilustración 17: contenido Audio Source. Fuente: SMPTE 306M

A continuación se describe la información de los bits que contiene el paquete AS.

PC0:

Este *byte* es el identificador con el valor 50 hex para el caso del paquete AS.

PC1:

LF: *flag* modo de bloqueo; indica si se ha bloqueado la frecuencia de muestreo de audio con la señal de vídeo

- 0 = modo bloqueado
- 1 = reservado

AF SIZE: número de muestras de audio por fotograma

- 0 1 0 1 0 0 b = 1600 muestras/cuadro (525/60 del sistema)
- 0 1 0 1 1 0 b = 1602 muestras/cuadro (525/60 del sistema)
- 0 1 1 0 0 0 b = 1920 muestras/cuadro (625/50 del sistema)

PC2:

CHN: el número de canales de audio dentro de un bloque de audio

- 0 0 b = un canal por bloque de audio
- Otros = reservado

El bloque de audio se compone de cinco sectores de audio en cinco pistas consecutivas para el sistema 525/60 y seis sectores de audio en seis pistas consecutivas para el sistema 625/50.

AUDIO MODE: el contenido de la señal de audio en cada canal

- 0 0 0 0 b = CH1
- 0 0 0 1 b = CH2
- 1 1 1 1 b = datos de audio no válidos
- Otros = reservado

PC3

50/60:

- 0 = 60-campo del sistema
- 1 = 50-campo del sistema

STYPE: define bloques de audio por fotograma de vídeo

- 0 0 0 0 0 b = 2 bloques de audio
- Otros = reservado

PC4

SMP: frecuencia de muestreo

- 0 0 0 b = 48 kHz
- Otros = reservado

QU: cuantificación

- 0 0 0 b = 16 bits lineal
- Otros = reservado

AAUX Source Control *pack* (ASC):

En la siguiente ilustración se muestra la estructura de los paquetes ASC.

	MSB							LSB
PC0	0	1	0	1	0	0	0	1
PC1	CGMS		Res	Res	Res	Res	EFC	
PC2	REC ST	REC END	FADE ST	FADE END	Res	Res	Res	Res
PC3	DRF	SPEED						
PC4	Res	Res	Res	Res	Res	Res	Res	Res

Ilustración 18: contenido Audio Source Control. Fuente: SMPTE 306M

A continuación, se describe la información de los bits que contienen el paquete ASC.

PC0:

Este *byte* es el identificador con el valor 51 hex para el caso del paquete ASC.

PC1:

CGMS: sistema de gestión de generación de copias

Posible generación de la copia:

- 0 0 = copia libre
- 0 1 = TBA
- 1 0 = TBA
- 1 1 = TBA

EFC: énfasis indicador de canal

- 0 0 b = énfasis *off*
- 0 1 b = énfasis *on*
- Otros = reservado

EFC se fijará para cada bloque de audio.

PC2:

EDIT ST: posición inicial de inserción de edición

- 0 0 b = parte sin editar
- 0 1 b = punto de edición sin desvanecimiento
- 1 0 b = punto de edición con desvanecimiento
- 1 1 b = reservado

La duración de la grabación EDIT ST será un período de bloque de audio para cada canal.

EDIT END: posición final de inserción de edición

- 0 0 b = parte sin editar

- 0 1 b = edición de punto sin desvanecimiento
- 1 0 B = edición de puntos con desvanecimiento
- 1 1 b = reservado

La duración de la grabación END EDIT será un período de bloque de audio para cada canal.

PC3:

DRF (*Interface*): sentido de la cinta

- 0 = sentido inverso
- 1 = sentido directo

SPEED (*Interface*): velocidad de cinta de VTR

SPEED	Shuttle speed of VTR	
	525/60 system	625/50 system
0000000	0/120 (=0)	0/100 (=0)
0000001	1/120	1/100
:	:	:
1100100	100/120	100/100 (=1)
:	:	Reserved
1111000	120/120 (=1)	Reserved
:	Reserved	Reserved
1111110	Reserved	Reserved
1111111	Data invalid	Data invalid

Ilustración 19: velocidad de cinta VTR. Fuente: SMPTE 306M

ANÁLISIS DE LOS FICHEROS DE AUDIO:

A continuación, se analizan los ficheros generados tras la descodificación que extrae la información del vídeo de prueba.

Los doce ficheros son llamados como audio_TT_FFF.dat, donde T indica el número de pista y F el campo de vídeo.

El contenido corresponde a los bloques de Audio Data y Audio Auxiliar Data, correspondientes a los datos del contenido del audio capturado y los datos auxiliares de audio que contiene los

metadatos, eliminando los datos de paridad y los bloques de sincronía. En la siguiente ilustración se muestran, marcados en rojo, los bloques de audio guardados en el fichero.

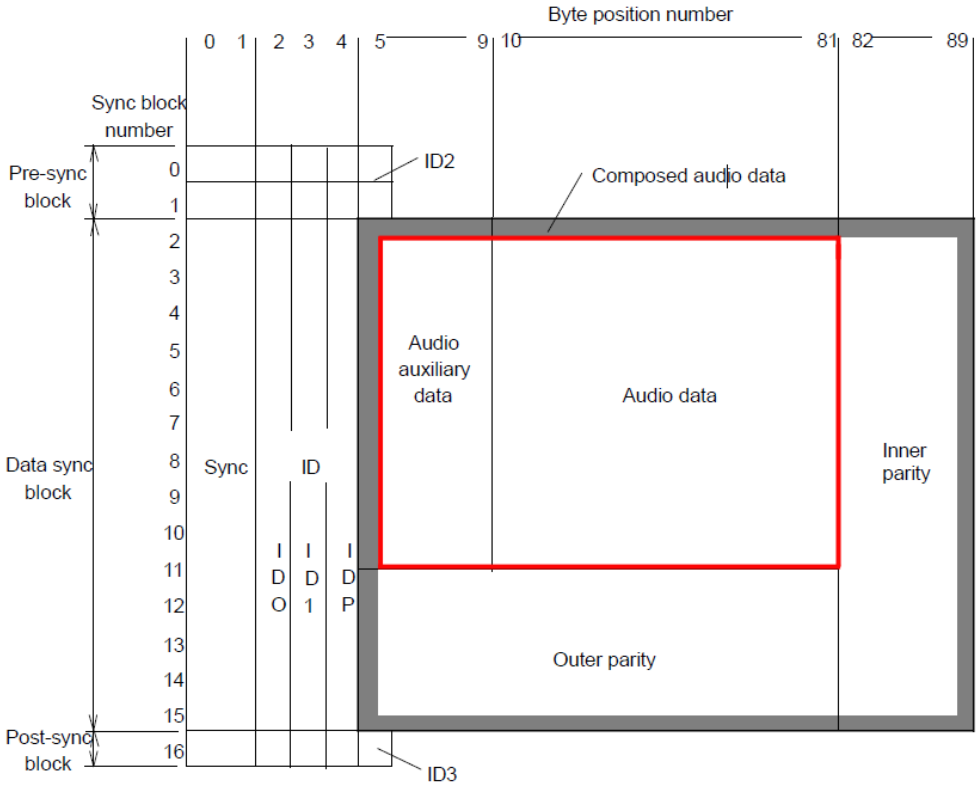


Ilustración 20: posición de Audio Auxiliar. Fuente: SMPTE 306M

Los ficheros contienen 9 filas de 77 bytes cada una. Los 5 primeros bytes de cada fila corresponden a los datos auxiliares de audio y los 68 restantes a los datos de audio. En el archivo de vídeo de prueba todos los bits del audio data tienen valor 0, por lo que corresponde a una señal constante de audio. Estos datos no son de interés para la funcionalidad buscada en el presente proyecto, causa por la que solo se analizarán los datos auxiliares.

6.3 *SUBCODE*

Los doce ficheros del sector *subcode*, llamados “subcode_DD_DDD.dat”, contienen los *bytes* de la cabecera ID (ID0, ID1 e IDP) y *subcode data* de las doce pistas, descartando los *bytes* de

detección de errores —Parity— y los de sincronismo —Sync— tal y como se representa en la siguiente figura:

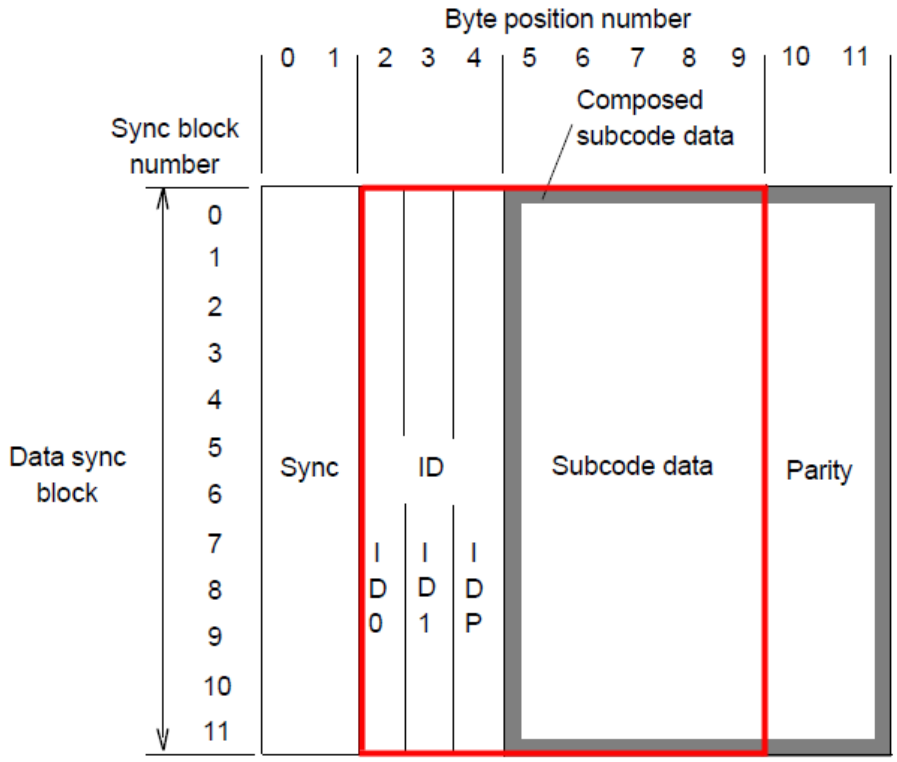


Ilustración 21: posición de *subcode*. Fuente: SMPTE 306M

Los *bytes* de interés para el análisis son los de la cabecera ID, guardando información de carácter general sin contener información relevante sobre una hipotética manipulación. En la siguiente tabla se muestra la correspondencia entre la posición de los bits y la información que contienen.

Bit position	Sync block numbers 0 and 6		Sync block numbers 1 to 5 and 7 to 10		Sync block number 11	
	ID0	ID1	ID0	ID1	ID0	ID1
b7 (MSB)	FR	Arb	FR	Arb	FR	Arb
b6	$AP3_2$	Arb	Res	Arb	APT_2	Arb
b5	$AP3_1$	Arb	Res	Arb	APT_1	Arb
b4	$AP3_0$	Arb	Res	Arb	APT_0	Arb
b3	Arb	Syb_3	Arb	Syb_3	Arb	Syb3
b2	Arb	Syb_2	Arb	Syb_2	Arb	Syb2
b1	Arb	Syb_1	Arb	Syb_1	Arb	Syb1
b0 (LSB)	Arb	Syb_0	Arb	Syb_0	Arb	Syb0

Ilustración 22: contenido *subcode*. Fuente: SMPTE 306M

Bastaría el análisis de uno de los 12 ficheros, ya que todos contienen la misma información en la cabecera, y de las filas, la 0 o 6, y la 11 del *byte* ID0, ya que el ID1 solo muestra el número del bloque de sincronismo representado por tres bits menos significativos.

En la siguiente ilustración se muestra el contenido de uno de los ficheros y la información que contiene:

146	ID0	ID1	IDP	3	4	SUBCODE DATA	6	7
0	10011000	00000000	00110101	00010011	00010011	00010011	00000111	00000000
1	11110001	00000001	11010111	00010100	00000000	00000000	00000000	00000000
2	11110000	00000010	11001001	00010011	00010011	00010011	00000111	00000000
3	11111000	00000011	01000100	00010011	00010011	00010011	00000111	00000000
4	11110001	00000100	11000110	00010100	00000000	00000000	00000000	00000000
5	11110000	00000101	11010010	00010011	00010011	00010011	00000111	00000000
6	10011000	00000110	00101011	00010011	00010011	00010011	00000111	00000000
7	11110001	00000111	11001001	00010100	00000000	00000000	00000000	00000000
8	11110000	00001000	11101011	00010011	00010011	00010011	00000111	00000000
9	11111000	00001001	01100110	00010011	00010011	00010011	00000111	00000000
10	11110001	00001010	11110000	00010100	00000000	00000000	00000000	00000000
11	10010000	00001011	10011010	00010011	00010011	00010011	00000111	00000000

Ilustración 23: contenido fichero *subcode*

El análisis del contenido, ayudándonos con una de las tablas, sería el siguiente:

Sync block 0: 10011000 FR= 1 $APR3_2$ =0, $APR3_1$ =0 $APR3_0$ =1

Sync block 11: 10011000 FR= 1 $APR3_2$ =0, $APR3_1$ =0 $APR3_0$ =1

En la siguiente figura se interpreta el valor de los bits APR:

Audio application ID			Format type
$AP3_2$	$AP3_1$	$AP3_0$	
0	0	0	Not used
0	0	1	D-7
0	1	0	Reserved
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	Not used

Ilustración 24: contenido APR

Como se puede observar, este sector solo informa del tipo de formato de la señal de vídeo, lo que ya extraíamos de *Stype* en Video Source. Por tanto, no merece la pena crear una pestaña

en la aplicación para mostrar la información de este sector, por lo que se ha descartado su análisis en el *software*.

6.4 *EDIT GAP*

Cada cuadro de televisión se graba en 10 pistas para el sistema 525/60, o 12 pistas para el sistema 625/50. Las pistas helicoidales contienen datos digitales en el sector ITI, el sector de vídeo, el de audio y el de subcódigo.

El espacio entre estos sectores en una pista es usado para informar sobre errores de tiempo durante la edición. En una grabación original, los *edit gap* se grabarán con la concatenación de los patrones de ejecución A o B definidos así:

MSB LSB

Patrón A: 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1

Patrón B: 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1 1 1 0 0

Durante una edición, los *edit gap* deben ser parcialmente reescritos con concatenaciones anteriores, suponiendo que los preámbulos y los epílogos de los sectores adyacentes se mantienen inéditos. Cada preámbulo de las zonas, excepto la 0, comienza con *run-up*. Cada epílogo de las zonas, excepto la 0, termina con la zona de guarda. Los *run-up* y las áreas de salva serán grabadas concatenadas con los patrones de ejecución A o B. Los patrones de ejecución A y B ya son patrones modulados bajo las reglas de entrelazado y modulación NRZI, que es donde se selecciona un modelo de ejecución entre A y B. La longitud de los huecos *edit gap* son:

- *Edit gap* 1: 625 bits

- *Edit gap* 2: 700 bits

- *Edit gap* 3: 1550 bits

7. METODOLOGÍAS DE TRABAJO

SCRUM es una metodología ágil para la gestión de proyectos donde se aplican, de manera regular, un conjunto de buenas prácticas para trabajar en equipo y obtener el mejor resultado posible en un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio sobre el trabajo de equipos altamente productivos (*The New Product Developement Game,* por Hirotaka Takeuchi (Hitotsubashi University) e Ikujiro Nonaka*).*

SCRUM no es una metodología para el desarrollo de aplicaciones, para eso hay otras metodologías ágiles, como *eXtreme Programming*, explicada más adelante, que serán integradas en el presente proyecto conjuntamente con SCRUM.

Las características principales son **[16]**:

- El desarrollo incremental de los requisitos del proyecto en bloques temporales cortos y fijos, en nuestro caso de un mes. Cada iteración, *sprint*, tiene que proporcionar un resultado completo, un incremento del *software* final que sea susceptible de ser entregado con el mínimo esfuerzo
- La priorización de los requisitos por valor y coste de desarrollo en cada iteración
- El control empírico del proyecto. Al final de cada iteración se muestra al tutor el resultado real obtenido, de manera que pueda tomar las decisiones necesarias en función de lo que observa y del contexto del proyecto en ese momento
- La sistematización de la colaboración y la comunicación con el tutor
- El *timeboxing* (todas las actividades tienen un tiempo máximo para conseguir unos objetivos) de las actividades del proyecto, para ayudar a la toma de decisiones y conseguir resultados

El proceso parte de la lista de objetivos/requisitos del *software*, que actúa como plan del proyecto. En esta lista el tutor prioriza los objetivos quedando repartidos en iteraciones y

entregas. De manera regular, el tutor puede maximizar la utilidad de lo que se desarrolla mediante la replanificación de objetivos que realiza al inicio de cada iteración.

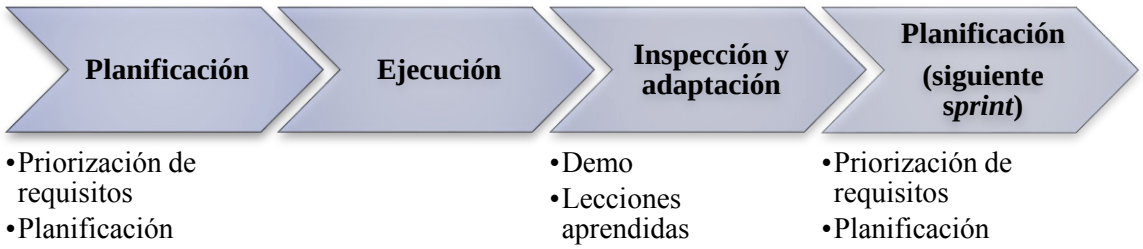


Ilustración 25: *sprint,* periodo de un mes

El primer día de cada *sprint* habrá una reunión con el tutor donde se priorizarán y planificarán los requisitos para ese mes. Durante la ejecución habrá reuniones con el tutor de corta duración para aclarar posibles dudas y el último día de esta iteración habrá una reunión para revisar los resultados y adaptarlos a nuevas necesidades.

Esta metodología está enfocada al mundo empresarial, donde los entornos de trabajo son complejos, donde se necesita obtener resultados pronto y los requisitos son cambiantes o poco definidos, donde la flexibilidad y la productividad son fundamentales. Al estar el proyecto solo compuesto por un integrante y como el "cliente" es el propio tutor, esta metodología ha sido adaptada y no se cumplirá con rigurosidad.

8. MODELO DE CICLOS DE VIDA DEL *SOFTWARE*

8.1 CICLO DE VIDA DEL SOFTWARE (ISO 12207)

Para definir el ciclo de vida de un *software* se han descrito una serie de modelos que establecen las fases y estados del *software* en el tiempo, especificando la forma en la que estas interaccionan entre sí. Han tomado tanta relevancia en la ingeniería de *software* que han sido reguladas por el estándar internacional ISO 12207.

Los procesos del ciclo de vida del *software* agrupan las actividades que se llevan a cabo durante el desarrollo de una aplicación.

Se agrupan en los tipos de procesos y actividades siguientes **[17]**:

- Procesos principales: definen las actividades llevadas a cabo por los actores principales que operan, desarrollan o mantienen el *software* durante su ciclo de vida. Los procesos principales son:
    - Proceso de adquisición
    - Proceso de suministro
    - Proceso de desarrollo
    - Proceso de operación
    - Proceso de mantenimiento
- Procesos de apoyo: definen las actividades auxiliares que dan apoyo a los procesos principales. Son:
    - Proceso de documentación
    - Proceso de gestión de configuración
    - Proceso de aseguramiento de la calidad
    - Proceso de verificación
    - Proceso de validación

    - Proceso de revisión conjunta
    - Proceso de auditoría
    - Proceso de solución de problemas
- Procesos organizativos: son aquellos llevados a cabo por organizaciones para la gestión y administración de los recursos y el personal que, en algún momento, participa en el ciclo de vida del *software*. Son:
    - Proceso de gestión
    - Proceso de infraestructura
    - Proceso de mejora del proceso
    - Proceso de recursos humanos

Buena parte de los procesos descritos no servirán para definir el modelo a seguir en el presente proyecto, ya que tiene una finalidad académica y dichos procesos están enfocados, en su mayoría, a estructuras empresariales.

En el caso de los procesos principales, el proceso de adquisición y el de suministro son llevados a cabo por el adquiriente (cliente) y proveedor respectivamente, por lo que no son actores tipificados en este proyecto. Del mismo modo, los procesos de operación y mantenimiento tampoco serán competencias del proyecto.

Los procesos organizativos tampoco serán contemplados al aplicar solo a aquellos proyectos en los que participan más de un analista, desarrollador o gestor.

Los procesos de apoyo serán considerados en la fase final del proyecto.

El proceso de desarrollo, dentro de los procesos principales, cumple con el fin que buscamos en este PFC. En él se describen las siguientes actividades:

1. Implementación del proceso
2. Análisis de los requerimiento del sistema
3. Diseño de la arquitectura del sistema
4. Análisis de los requerimientos del *software*
5. Diseño de arquitectura del *software*
6. Codificación y pruebas del *software*
7. Integración del *software*

8. Pruebas de calificación del *software*
9. Integración del sistema
10. Pruebas de calificación del sistema
11. Instalación del *software*
12. Apoyo de aceptación del *software*

En el presente proyecto se sintetizarán los procesos aceptados anteriormente, diferenciando entre las siguientes fases:

- Definición de requisitos: es la fase más temprana del ciclo de vida de una aplicación, en la que se analizan los objetivos que definen la finalidad del desarrollo y las funcionalidades que dan solución a las necesidades que debe cubrir el *software*. En esta fase se redacta el EDRF (Especificación Detallada de los Requisitos Funcionales), donde se describen los casos de uso, y el EIU (Especificación de Interfaz de Usuario), que describe los componentes de las diferentes pantallas interactivas. Una vez se redactan estos documentos, con la información que estos aportan, se realiza la estimación temporal, permitiendo la planificación y la agrupación de las funcionalidades en subconjuntos
- Diseño técnico: en esta fase se utilizan leguajes como el UML, que permiten a los creadores "analistas", que han definido los requisitos, comunicar al programador las directrices que le permitan abordar el desarrollo tal y como se ha definido en la fase previa
- Desarrollo: es la fase en la que se implementa todo lo que se ha definido en el diseño técnico, siendo aquella en la que más tiempo se emplea
- Plan de pruebas: comienza con el diseño de un plan de pruebas diferenciando entre los tipos de pruebas. Estas pueden ser de los siguientes tipos:

  Niveles de pruebas:

  - Pruebas unitarias
  - Pruebas integrales

  Pruebas funcionales:

  - Pruebas regresivas

- Pruebas funcionales
- Pruebas de aceptación

Pruebas no funcionales:

- Pruebas de rendimiento
- Pruebas de seguridad
- Pruebas de portabilidad
- Pruebas de mantenibilidad

Validación del usuario y documentación:

La validación de usuario, como ya se explicó anteriormente, no va a poder realizarse antes de la presentación del proyecto. En cuanto a la documentación, además del presente documento se redactará y se adjuntará la que sigue:

- EDRF (Especificaciones Detalladas de Requisitos Funcionales)
- EIU (Especificaciones de Interfaz de Usuario)
- Diseño técnico
- Guía de usuario
- Plan de pruebas

8.2MODELOS DE CICLOS DE VIDA DEL *SOFTWARE*

MÉTODOS CLÁSICOS

Metodología en cascada (Royce) [18]:

Es uno de los llamados métodos clásicos, al ser el primero que se definió para el desarrollo de aplicaciones informáticas. En la actualidad, es un método muy discutido tanto en el ámbito académico como profesional; aun así, sigue siendo el más extendido.

Con este método, los requisitos se fijan de forma predeterminada al comienzo del ciclo de vida del desarrollo. El inconveniente que esto conlleva es cuando, posteriormente, durante el

desarrollo y, de forma más notable, en la fase de pruebas, se encuentra la necesidad de volver a redefinir nuevas funcionalidades o corregir errores. Esta falta de flexibilidad hace que no sea el método más idóneo, teniendo en cuenta que la ingeniería de *software* es un contexto lleno de incertidumbre y riesgos.

El ciclo de vida en el desarrollo de *software* que antes hemos descrito se puede sintetizar en las siguientes fases:

Ilustración 26: modelo en cascada

Esta disposición lineal e ideal de las metodologías en cascada se convierte, al final, en una realimentación entre las fases finales y previas, lo que se traduce en costes extraordinarios que no habían sido considerados en un principio.

MÉTODOS EVOLUTIVOS

Metodología espiral o RUP [19]:

Es uno de los llamados métodos evolutivos; fue ideado inicialmente por B. Boehm en 1998 y se caracteriza por la aparición del análisis de riesgos, que permite evaluar la viabilidad de un proyecto. Se puede considerar como una espiral en la que cada ciclo es una fase del modelo clásico en cascada, por lo que puede entenderse como una solución intermedia entre el modelo en cascada y el modelo de prototipos.

Cada ciclo se divide en cuatro sectores:

- Determinar objetivos: se estudia la finalidad de la fase del ciclo de vida en la que se encuentre el proyecto. Se identifican las debilidades del sistema *software* y los riesgos, y se planifica el ciclo
- Análisis del riesgo: se realiza un análisis de los riesgos identificados y los planes de contingencia en los que se detallan estrategias alternativas

- Desarrollar y validar: en este sector de la elipse es cuando se implementa la opción que haya sido identificada como más viable
- Planificación: Una vez terminado y tras ser revisado por el usuario, se decide si se pasa a otro ciclo, planificándolo, o si se vuelve a iterar el mismo ciclo

Las diferentes versiones de la aplicación nacen de las iteraciones con la espiral.

Modelo iterativo incremental [20]:

Es uno de los llamados métodos evolutivos, caracterizados por ser modelos eficientes en aquellos casos donde no están del todo definidos los requisitos por parte del usuario.

Los requisitos se agrupan en subconjuntos por funcionalidades, describiendo así los incrementos separados temporalmente por un "desajuste". Los primeros incrementos que se desarrollan son los que aportan las funcionalidades principales y mejor definidas. Cada incremento se desarrolla, a su vez, como un desarrollo en cascada.

Esta disposición temporal de los incrementos permite dotar a los usuarios de las principales funcionalidades en las primeras versiones del *software*. Este *feedback* del cliente con el *software* aporta la información necesaria para definir nuevas funcionalidades y especificar aquellas que no estaban del todo definidas.

En el siguiente gráfico se muestra un ejemplo del modelo y la disposición temporal de los incrementos:

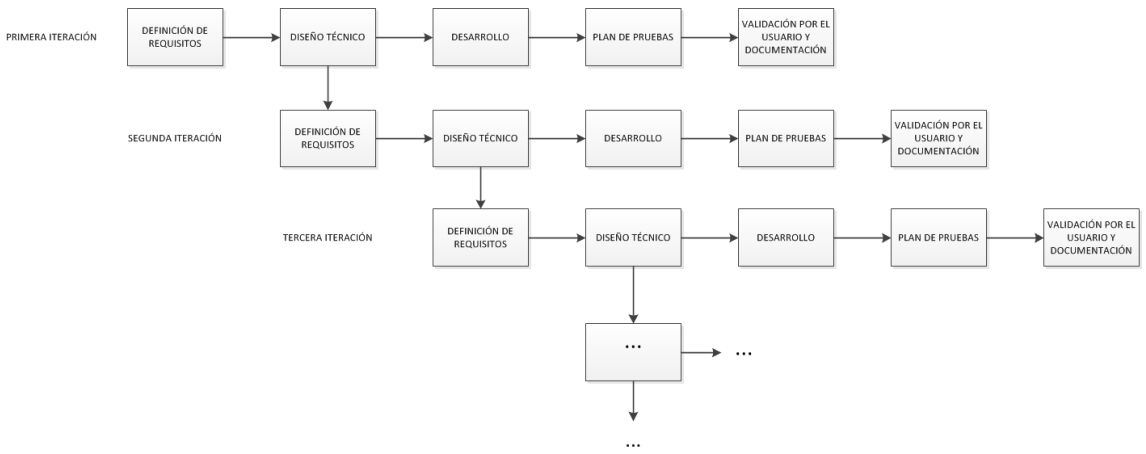


Ilustración 27: modelo iterativo incremental

Modelo de prototipos [21]:

Un prototipo es una aproximación experimental de una solución final, en este caso del *software*. Este modelo se basa en el desarrollo rápido de un prototipo cuya evolución se realimenta de las interacciones con el usuario final, el cual, tras su experiencia, aporta la información necesaria para definir nuevos requisitos. Este modelo muestra ventajas si el objetivo final de la aplicación está bien definido pero no así los parámetros de entrada y los resultados de salida.

Se pueden identificar los siguientes modelos de prototipos:

- Prototipos evolutivos: son aquellos que sirven de base para desarrollar sobre ellos la solución final. Esto conlleva una serie de precondiciones que se describen a continuación:
    - La tecnología utilizada para el desarrollo del prototipo debe ser la misma que la de la aplicación final
    - Que estén bien definidos los requisitos principales
    - Evaluaciones por parte del usuario que permiten definir la evolución y refinamiento del prototipo hasta la solución final
- Prototipos desechables: en este caso, el prototipo es desechado cuando cumple su función, caracterizándose por ser una versión de baja calidad. Se caracteriza por:
    - No usar, por norma, el mismo lenguaje y método de programación que la aplicación final, buscando siempre la rapidez y facilidad en el desarrollo del prototipo
    - Ser utilizado en caso de aplicaciones con factores críticos
- Combinación de prototipos evolutivos y desechables: esta combinación se estructura de tal forma que las partes que están bien definidas se desarrollan como prototipos evolutivos, siguiendo un enfoque de implementación riguroso, y aquellas que no están aún definidas y que son susceptibles de drásticos cambios lo hacen como prototipos desechables, donde se prioriza la rapidez y no la rigurosidad

Metodología ágil [22]:

La programación extrema o *eXtreme Programming* (XP) es un enfoque de la ingeniería de *software* formulado por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change* (1999).

Es el más destacado de los procesos ágiles de desarrollo de *software*. Al igual que estos, la programación extrema se diferencia de las metodologías tradicionales, principalmente, en que pone más énfasis en la adaptabilidad que en la previsibilidad.

Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los mismos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del *software*.

Las características fundamentales del método son:

- Desarrollo iterativo e incremental: pequeñas mejoras, unas tras otras
- Integración continua: se generan los ejecutables de manera automática y se repiten de forma automatizada (cuando se pueda) las pruebas unitarias, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación
- Corrección de todos los errores antes de añadir una nueva funcionalidad, lo que conlleva hacer entregas frecuentes
- Refactorización del código, es decir, rescribir ciertas partes del mismo para aumentar su legibilidad y mantenimiento sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo
- Propiedad del código compartido: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados

- Simplicidad en el código: es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo
- Minimizar la documentación. Presentación con los requerimientos, documentación de pruebas, manual de usuario y código comentado

9. DEFINICIÓN DE REQUISITOS (ANÁLISIS FUNCIONAL)

En este apartado se tratarán de describir, brevemente, los documentos que componen el análisis funcional en un proyecto profesional de *software*.

Esta fase dentro del ciclo de vida del *software* es en la que el usuario plantea el problema y, junto al analista, se detallan cuáles van a ser las especificaciones que va a tener el sistema. Los documentos asociados al presente documento y que definen esta fase son el EDRF (Especificación Detallada de Requisitos Funcionales) y el EIU (Especificaciones de Interfaz de Usuario).

9.1 EDRF

El análisis de requisitos es la primera de las fases de un ciclo de desarrollo de *software*, donde se definen las características del *software* que den solución a las necesidades del usuario. Es la fase en la que la comunicación entre usuario y analista se hace más importante, por lo que saber trasladar al cliente las posibles soluciones técnicas a su problema y ayudar a que él mismo defina la lógica es uno de los objetivos de esta fase y de la que depende gran parte del éxito de cualquier proyecto de *software*.

El documento donde se recoge el análisis de los requisitos funcionales es el llamado EDRF (Especificación Detallada de Requisitos Funcionales), en el que se describen las características de la aplicación por medio de los casos de uso.

Un caso de uso es la representación de una funcionalidad como una secuencia de acciones del actor y respuesta del sistema, siendo el actor la tipificación del rol del usuario. **[23]**

Los casos de uso se estructuran como sigue:

- Nombre: nombre del caso de uso, debe definir con claridad y de forma concisa la funcionalidad que va a describir
- Descripción: descripción breve de la funcionalidad
- Precondiciones: condiciones previas que se deben dar para que se dé la funcionalidad
- Resultado: condiciones posteriores al terminar el caso de uso
- Evento que inicia el caso de uso: es el evento que tiene que darse para que comience el caso de uso. Esto nos permite no tener que describir aquellos pasos previos que no añaden valor para el caso de uso que se describe. Por ejemplo, en el caso de uso de modificar el coste de una operación de un hipotético almacén, no haría falta especificar todas las interacciones entre el actor y el sistema que ya se describen en otros casos de uso, como puede ser, en este caso, la búsqueda de la operación, sino que bastaría con referenciar el caso de uso precedente en este campo
- Requisitos no funcionales: son aquellos requisitos del sistema que son necesarios para que se pueda dar el caso de uso, como puede ser un *hardware* específico (una *webcam* como requisito no funcional para la funcionalidad: por ejemplo, realizar la conexión con videoconferencia)

9.2 EIU

Es el documento en el que se describen los componentes que contiene una pantalla concreta. Suelen ser estructurados con la descripción de cada componente, siempre tratando de no ser muy técnicos en la explicación para facilitar su entendimiento por parte del usuario, y acompañándolo de una captura de pantalla.

En caso de que todavía no se haya desarrollado la pantalla, se sustituirán las capturas de la aplicación por prototipos que sean lo más parecidos posible al resultado final sin necesidad de llegar a un nivel alto de detalle desde el punto de vista del diseño gráfico.

10. DISEÑO TÉCNICO

La falta de un convenio a la hora de plantear el diseño técnico con una notación normalizada que permita describir por medio de modelos, funcionalidades, clases, objetos y secuencias hizo que algunos ingenieros de *software* trabajasen en busca de un lenguaje estandarizado. En 1997, el consorcio formado por las compañías DEC, Hewlett-Packard, Intellicorp, Microsoft, Oracle, Texas Instruments y Rational, y tomando como referencia las investigaciones de James Rumbaugh, Grady Booch e Ivar Jacobson, desarrollaron la versión 1.0 del lenguaje UML (Lenguaje de Modelado Unificado).

El lenguaje UML permite, mediante grafos interrelacionados, representar una abstracción de la realidad del problema por medio del diseño de un modelo. Estos grafos son de fácil interpretación tanto para desarrolladores como para el usuario final, lo que tiende un puente de entendimiento entre ambas partes en este aspecto que nunca está exento de conflictos. El hecho de simplificar el sistema por medio de un modelo independiente a la implementación permite que un mismo diseño técnico pueda ser desarrollado con cualquier tecnología que cumpla unos requisitos, por lo que el analista que realiza la toma de requisitos y el diseño técnico no tiene por qué conocer la tecnología de implementación, lo que permite independizar las funciones de ambos actores en sus respectivas competencias en el ciclo de vida del *software* **[24].**

Estos grafos se interrelacionan formando diagramas de diferentes tipos; a continuación, se sintetizarán aquellos que tendremos en cuenta en el presente proyecto.

Diagrama de casos de uso [25]:

Los casos de uso, como ya se definió en la sección de análisis funcional, describen con detalle una funcionalidad por medio de las relaciones que hay entre las acciones que realiza el usuario (actor) en el sistema por medio de su interfaz y la respuesta de este tras la interacción.

Las relaciones entre los casos de uso pueden ser de:

- Asociación: es la relación más básica, representando la invocación de un actor a un caso de uso. La representación gráfica de esta relación es por medio de una línea continua

- Inclusión: es cuando un caso de uso forma parte de otro teniendo en común algunos pasos. Por lo tanto, el caso de uso inicial incluye la funcionalidad del caso de uso final. Esta relación es representada por líneas discontinuas etiquetándose con la palabra <<*include*>>

- Extensión: es la relación en la que un caso de uso está basado en otro más básico, siendo, por tanto, una especialización de un caso más genérico. La representación es igual que la anterior pero con la etiqueta <<*extend*>>

- Generalización: es el caso en que un caso de uso (subcaso) hereda de otro caso de uso (primario) su comportamiento, las relaciones descritas anteriormente y su significado

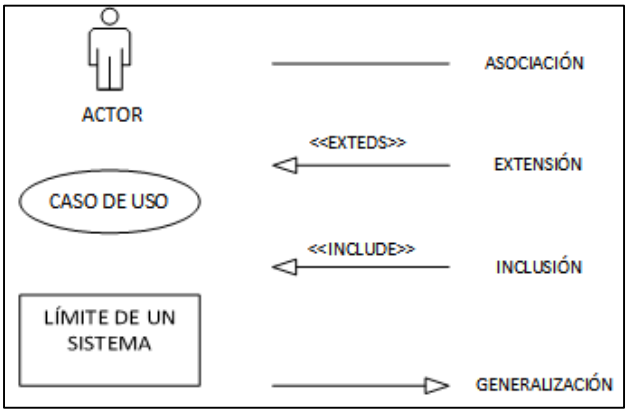


Ilustración 28: grafos en diagramas de casos de uso

Diagrama de clases [26]:

Las clases están representadas por una serie de atributos y operaciones, siendo los diagramas de clases las relaciones entre diferentes clases.

La representación de una clase, en un diagrama de clases, se estructura en tres rectángulos que, de arriba abajo, indican el nombre de la clase, los atributos y las operaciones,

respectivamente. Tanto en los atributos como las operaciones se indica el nivel de accesibilidad representado como:

- + *public*: método visible tanto dentro como fuera de la clase
- - *private*: indica que solo es accesible desde dentro de la clase
- # *protected*: indica que no será accesible desde fuera de la clase pero sí por los métodos de esta y, además, por sus subclases (herencia)

Las relaciones entre clases se representan por líneas que unen unos conjuntos de rectángulos con otros y cuyos tipos se diferencian y se representan como sigue:

- Herencia: indica que una subclase hereda de una clase principal sus atributos y métodos que sean *public* y *protected*. El sentido de la flecha apuntará a la clase principal
- Agregación: sirven para modelar instancias de clases, diferenciándose en el tiempo de vida del objeto respecto a la clase instanciada. Los tipos de agregación son:
    - Por valor: es un tipo de relación estática, en la que el tiempo de vida del objeto incluido está condicionado por la clase principal. Se representa con un rombo con relleno
    - Por referencia: es un tipo de relación dinámica donde el tiempo de vida del objeto incluido es independiente. Se representa con el rombo sin relleno
- Asociación: permiten asociar clases entre sí, indicándose en sus extremos la multiplicidad que indica el grado y el nivel de dependencia
    - Estas pueden ser:
        - **1** Un elemento relacionado
        - **0..**1Uno o ningún elemento relacionado
        - **0..*** Varios elementos relacionados o ninguno
        - **1..*** Varios elementos relacionados pero al menos uno
        - ***** Varios elementos relacionados

- **M..N** Entre M y N elementos relacionados
- **Dependencia:** indica la instanciación de clases, representada con una flecha discontinua que va desde el objeto hasta la clase

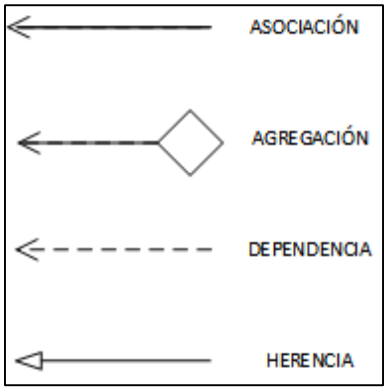


Ilustración 29: grafos en diagramas de clases

Diagrama de secuencia [27]:

Es el tipo de diagrama al que más énfasis se dará en el presente proyecto a la hora de plantear el diseño técnico. En este se representarán las clases que componen el programa y las llamadas que se hacen para realizar una tarea determinada, por lo que por cada caso de uso del EDRF tendremos un diagrama de este tipo. Estos diagramas están representados en dos dimensiones, la horizontal, donde se representa la disposición de los objetos, y la vertical, en la que se representa el tiempo.

La representación de los objetos y sus relaciones es como se detalla a continuación:

- **Clases:** se representan con un rectángulo etiquetado en su interior con el nombre de la clase subrayado. Las clases se colocan de forma correlativa en la parte superior y de izquierda a derecha en función de su posición dentro de la secuencia de acciones
- **Línea de vida:** se representa con líneas discontinuas extendidas en vertical debajo de las clases y en las que se representan con rectángulos las operaciones que se realizan sobre el objeto
- **Mensajes:** se representan con flechas que van desde unas líneas de vida a otras y que son etiquetadas con el método de la clase que es llamado desde la clase precedente o

por el actor por medio de un evento consecuencia de una acción en un componente de la IU. Este tipo de mensajes pueden ser de varios tipos:

  - **Simples**: representan la transferencia de control entre clases
  - **Síncronos**: es cuando el sistema espera la respuesta de este mensaje para continuar
  - **Asíncronos**: es cuando el sistema no espera la respuesta de este mensaje para continuar

- Como se indicó anteriormente, la línea vertical representa el paso del tiempo, por lo tanto mientras la flecha que representa un mensaje está en un nivel superior a otra, este mensaje ocurre antes que el otro.

También hay que hacer mención a que, en ciertas ocasiones, las llamadas se hacen desde y hasta la misma clase desde donde se realizan. Este tipo de mensajes recursivos se representan con una línea cerrada que empieza y termina en la misma línea de vida.

11. DESARROLLO

En este apartado no se pretende detallar cómo se ha desarrollado la aplicación y cómo se ha configurado el proyecto para cumplir con los objetivos perseguidos con el *software*.

En el documento adjunto "Diseño Técnico" es donde se exponen las clases, configuraciones y, en definitiva, las peculiaridades del desarrollo de la aplicación. Por tanto, en este apartado solo se hará una introducción a la tecnología MFC y al tipo de proyecto que se ha creado finalmente.

Además, se describirá la plataforma de desarrollo y el *software* que se ha utilizado para poder implementar el *software*.

11.1 PLATAFORMA DE DESARROLLO

A pesar de ser un proyecto fin de carrera —por lo que, en consecuencia, solo un desarrollador va a programar—, se ha visto conveniente diseñar y configurar una plataforma que permita al tutor una supervisión y seguimiento de la evolución de la aplicación y de la documentación asociada.

En este apartado se describe la solución a la que se ha llegado para satisfacer esta expectativa, describiendo los datos técnicos de diseño de la plataforma y su configuración. Se describe también todo el *software* que será utilizado en el proyecto, explicando cuál es la funcionalidad con la que cumple dentro del mismo.

CVS (SISTEMA DE CONTROL DE VERSIONES)

Los sistemas de control de versiones se han extendido en el ámbito del desarrollo profesional de *software*, siendo en la actualidad una herramienta fundamental usada conjuntamente con los entornos de desarrollo integrado. La principal función con la que cumplen estos sistemas

es permitir a varios programadores trabajar en un mismo desarrollo, teniendo archivadas las versiones anteriores que sean estables.

Los CVS (Sistemas de control de versiones) tienen una arquitectura cliente-servidor. El servidor es donde se guarda el repositorio con la última versión de *software* en la que se desarrolla y las versiones estables anteriores, mientras que el cliente es una aplicación que permite conectar con el servidor con las funcionalidades de importar, exportar, etiquetar, identificar, eliminar y sustituir los archivos que se encuentran en el repositorio.

El lado servidor podría implementarse con un servidor http como Apache. El inconveniente es tener el *host* conectado para dar un servicio 24/7, además de la necesidad de tener una conexión *full-duplex* para un correcto funcionamiento.

Como alternativa a las limitaciones de infraestructura, se ha visto conveniente optar por los sistemas de computación en la nube, por lo que se ha instalado el CVS en el sistema de alojamiento de archivos multiplataforma en la nube, Dropbox.

El CVS que se va a instalar es Subversión, siendo la aplicación cliente TortoiseSVN. Para la instalación se deben seguir los siguientes pasos:

1. Registro de cuenta e instalación de Dropbox (servidor). www.dropbox.com

2. Instalación de Tortoise (usuario). www.tortoisesvn.net

3. Tras eliminar todas las carpetas creadas de forma predeterminada en el alojamiento de Dropbox, se crea la carpeta donde se va a alojar el repositorio. Sobre la carpeta se hace clic con el botón derecho del ratón y se selecciona la opción "*Create repository here".* Aparece un cuadro de diálogo y se hace clic sobre la pestaña "*Create folder structure*" para crear las tres carpetas con las que se suele estructurar un CVS.

4. Para poder acceder a la ventana principal de gestión de la SVN, se hace clic con el botón derecho sobre el escritorio y se accede a TortoiseSVN /Repo-broser.

5. Se han creado tres carpetas:

- Branches: es donde se alojan los desarrollos que se estén programando de forma paralela por varios programadores

- Tags: es donde se guardan las versiones previas que son estables
- Trunk: es la carpeta raíz, en la que se encuentran los archivos con los que se estén trabajando durante el desarrollo de la nueva versión

6. Para subir los archivos al repositorio pulsamos en la carpeta correspondiente y hacemos clic en el ítem" *Add file"*.

En las carpetas Branches y Tags se han importado las carpetas SAVID v6, que es la versión de inicio, y en Trunk la nueva versión SAVID v7.

A continuación se explica cómo trabajar con el CVS.

Guía rápida de usuario:

1. Se crea una carpeta en nuestro terminal en la que se vuelca el repositorio y donde se trabaja en entorno local.

2. Estando en modo de sincronización con DropBox, se coloca el cursor sobre la carpeta destino y se pulsa el botón derecho del ratón, seleccionando la opción de "*Checkout*". Se abre una ventana de diálogo que contiene una caja de texto donde se escribe la ruta en la que se encuentra la carpeta que sincroniza con Dropbox. La ruta se inicia con el prefijo "file:///"

3.Una vez se baje todo el proyecto a la carpeta local, las acciones más comunes que se realizan con el cliente TortoiseSVN son las de *commit*, para subir todos los cambios que hayan realizado en nuestro host, y *update*, para bajar los archivos modificados y subidos por el resto de desarrolladores. Es una buena práctica el realizar siempre un *update* antes de realizar un *commit* para asegurarnos de no crear conflictos. Estos conflictos se producen cuando dos desarrolladores paralelamente han tocado las mismas líneas de código; en ese caso se debe elegir entre mantener nuestro código o sustituirlo por el del otro programador.

IDE (ENTORNO DE DESARROLLO INTEGRADO)

El IDE que se ha utilizado para el desarrollo de la aplicación es Microsoft Visual Studio 2010 Profesional, que permite programar aplicaciones para la plataforma Windows. Soporta un amplio número de lenguajes de programación, entre ellos C++, que es el lenguaje en que están desarrolladas las clases MFC, siendo esta la tecnología con la que se ha programado la interfaz

gráfica. Como toda IDE, MVS está compuesta por un editor de textos, un compilador, un depurador y una herramienta para la creación de interfaz gráfica (GUI).

El proyecto es de tipo MFC (cuadro de diálogo), por lo que de manera predeterminada, en función de la configuración seleccionada en el *wizard*, se crearán y referenciarán las clases MFC específicas para este tipo de proyecto. El IDE tiene una herramienta de autocompletado llamada *Intellisense*, lo que permite un desarrollo más rápido y eficiente. Una peculiaridad es que, en caso de que en la configuración del proyecto se seleccione compatibilidad con CLR, se permitirá referenciar clases del framework .NET que, previamente, hayamos incluido en el proyecto.

El depurador, *debugger,* nos permite, por medio de puntos de interrupción, navegar por el código y ver el valor que van tomando las variables, el acceso dentro de las estructuras condicionales, los puntos donde se producen excepciones o los *warning* que no han sido corregidos.

Esta IDE tiene un CVS evolucionado llamado TFS, Team Foundation System que, por su complejidad, no se ha visto necesario utilizar.

La Universidad Politécnica de Madrid y Microsoft tienen un convenio por el que sus alumnos son beneficiarios de una suscripción a MSDN (MicroSoft Developer Network), donde se pueden bajar las herramientas de desarrollo de aplicaciones en la plataforma Microsoft con sus respectivas licencias. Por este motivo, se ha respetado rigurosamente el cumplimiento de la legislación de *software* bajo licencia.

***SOFTWARE* COMPLEMENTARIO**

Microsoft Project 2010: es una aplicación creada por Microsoft, pensada para la planificación y administración de proyectos que se ha extendido ampliamente en el ámbito profesional. Esta herramienta permite a cualquier administrador hacer seguimiento en el transcurso del tiempo del cumplimiento de objetivos, desarrollar planes de trabajo y analizar cargas de trabajo.

Visio Premium 2010: es una aplicación, también bajo licencia de Microsoft, que permite diseñar diagramas de una forma rápida e intuitiva. De forma predeterminada se puede seleccionar un proyecto de algún tipo de diagrama cuya GUI incluye grafos y conectores

específicos para ese tipo. En el proyecto, además de algunos diagramas explicativos que complementan el texto, se ha utilizado para realizar el diseño técnico del *software* con sus correspondientes diagramas de clases y diagramas de secuencia.

Dropbox: es un servicio de alojamiento de computación en la nube. Permite al usuario guardar, sincronizar y compartir archivos con todos los usuarios. Como se comentó anteriormente, la idea es que el tutor tenga acceso al repositorio de todos los archivos de desarrollo y documentación, para llevar un seguimiento en tiempo real del proyecto.

Subversion: conocido como SVN, es un sistema de control de versiones de licencia libre tipo Apache/BSD. Este tipo de *software* permite automatizar las tareas de guardar, recuperar y registrar versiones de desarrollo de aplicaciones. La razón por la que se va a utilizar en el proyecto es más académica que puramente práctica, ya que se tratará de seguir un paralelismo con los entornos de trabajo en el mundo profesional.

11.2 MFC

En 1992, Microsoft presentó MFC, que son una serie de clases interrelacionadas que permiten al usuario un acceso más eficiente a la API de Windows para la creación de aplicaciones en esta plataforma, siendo el lenguaje de programación C++.

En muchas funciones se utiliza el prefijo Afx debido a que el nombre MFC se adoptó tardíamente y, en un principio, se llamaba Extensiones de Application Framework. De una manera predeterminada todos los proyectos MFC tienen el estándar de encabezado pre compilado stadfx.h.

En las aplicaciones MFC, como SDI y MDI, podemos diferenciar entre la información (documento), la forma en que esta se muestra al usuario (vista) y el contenedor de este binomio (marco), en lo que se llama arquitectura documento-vista.

El asistente de aplicaciones *wizard* crea de forma predeterminada una clase documento y otra vista.

Por un lado, los documentos contienen los datos de diferentes formatos, desde una base de datos a un documento de texto como es el caso de este proyecto. Además de guardar los

datos, también administra la impresión de los mismos y coordina la actualización de las vistas. Un objeto de documento en MFC nos permite leer y escribir datos en un dispositivo de almacenamiento. Por lo tanto, los cambios en los datos que puede realizar el usuario por medio de las vistas son administrados mediante esta clase.

Las vistas muestran los datos permitiendo la interactuación con el usuario. Un objeto-vista en MFC administra la presentación de los datos, desde el procesamiento de los mismos en una ventana, a su selección y edición por parte del usuario.

Seguir este tipo de arquitectura tiene como beneficios mostrar distintos tipos de documentos, un mismo documento con diferentes vistas y la posibilidad de crear ventanas divisoras.

Las clases más importantes en un proyecto MFC con esta arquitectura son:

CDocument: es la que nos da funcionalidad para clases de documentos definidos por el desarrollador. Nos permite, fundamentalmente, crear, cargar y guardar archivos. La interfaz definida por esta clase es utilizada por el marco para manipular documentos.

Imaginemos una aplicación que pueda mostrarnos diferentes tipos de documentos, como puede ser un Excell y un archivo de texto. Al ser dos archivos diferentes necesitarán plantillas diferentes que contengan la barra de menú del marco adecuado. La gestión de recursos que se usa para mostrar un marco diferente para cada tipo de documento la lleva a cabo esta clase. Así, cada tipo de documento tendrá un puntero asociado al objeto CDocTemplate.

Si modificamos un documento por medio de la interacción con Cview, clase que se explica a continuación, las vistas deben reflejar estos cambios. La función UpdateAllViews es la que notifica estos cambios para refrescar la vista del documento. El marco siempre pedirá al usuario guardar un archivo modificado antes de cerrarlo.

CView: es la que nos da funcionalidad para clases de vistas definidas por el desarrollador. La vista sirve de intermediaria entre el documento y el usuario, procesando una imagen del documento en pantalla y registrando peticiones de cambio por parte del usuario sobre el documento.

CFrameWnd: representa el marco contenedor que contiene una o varias vistas del documento.

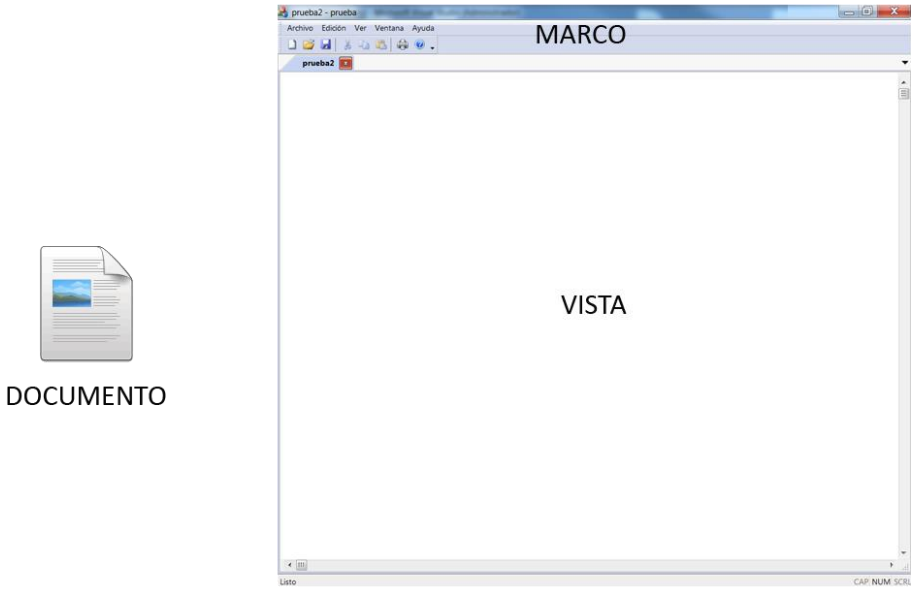


Ilustración 30: arquitectura documento-vista

Tipos de proyecto con arquitectura documento-vista:

- SDI (*Single Document Interface*): aplicaciones de un solo documento
- MDI (*Multiple Document Interface*): aplicaciones que permiten múltiples marcos de documento abiertas. Por tanto, es una ventana principal en la que se pueden abrir ventanas secundarias

Una vez vista la arquitectura típica de los proyectos MFC, se deduce que no cumple con las características idóneas para mostrar la información.

Los ficheros creados por SAVID Procesador no son más que código binario, por lo que de poco serviría crear una aplicación que los abriese sin procesar este código y mostrar la información subyacente.

Así pues, aprovechando que MFC da la alternativa de crear un proyecto “cuadro de diálogo”, aunque con ciertas limitaciones, y la posibilidad de compatibilidad con CLR, se ha optado por esta opción. De esta manera no renunciamos a utilizar esta API libre y estructuramos la aplicación de un modo adecuado, posibilitando la arquitectura en tres capas.

La información de esta sección se ha obtenido de MSDN referenciado en **[28]** de la bibliografía.

11.3 DESCRIPCIÓN DE LAS CLASES DEL PROYECTO SOFTWARE

Con el objetivo de describir las clases de la aplicación estas se han estructurado en tablas cuyas celdas se detallan a continuación.

Encabezado: en el lenguaje de programación C++ son los archivos de descripción de interface, donde se declaran las variables, los métodos y las directivas de preprocesador. Los archivos tienen una extensión .h y en el proyecto MFC se encuentran en la carpeta de "Archivos de encabezado".

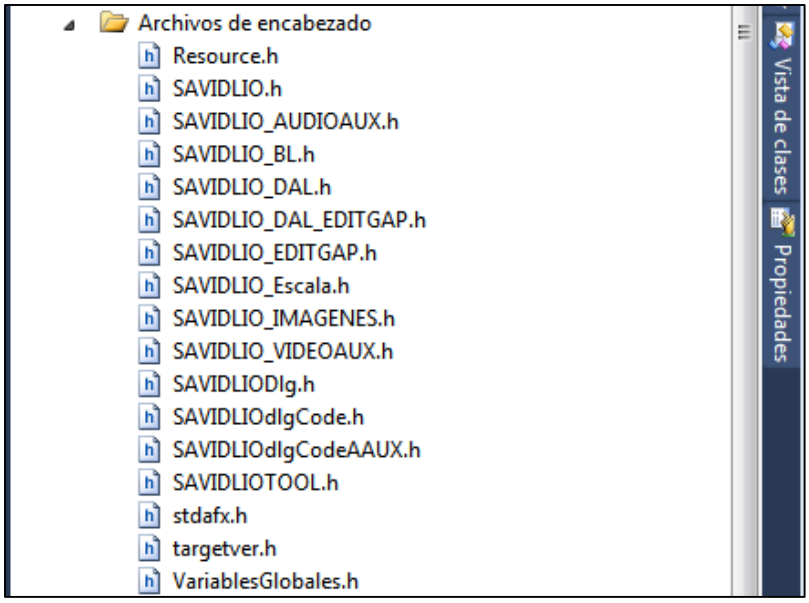


Ilustración 31: archivos de encabezado. "Explorador de soluciones"

Preprocesador de C (CPP): en el lenguaje de programación C++ son los archivos que contienen el código fuente que implementan las funcionalidades de la aplicación. Los archivos tienen una extensión .cpp y en el proyecto MFC se encuentran en la carpeta de "Archivos de Código Fuente".

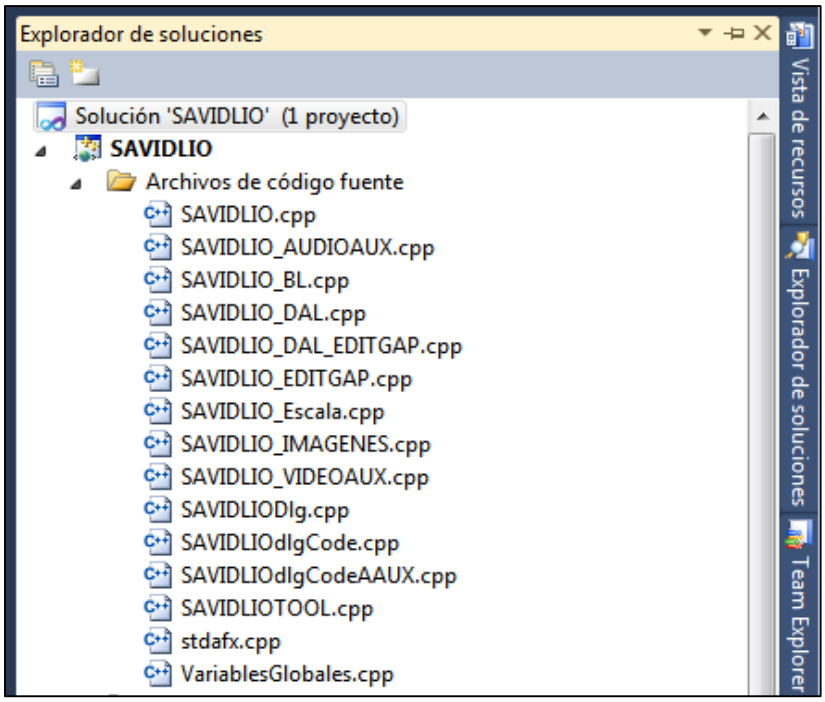


Ilustración 32: archivos de código fuente. "Explorador de soluciones"

Recurso: indica el recurso de tipo cuadro de diálogo al que está asociado la clase. En la aplicación se encuentra en la carpeta "*Dialog*" dentro de "Vista de recursos".

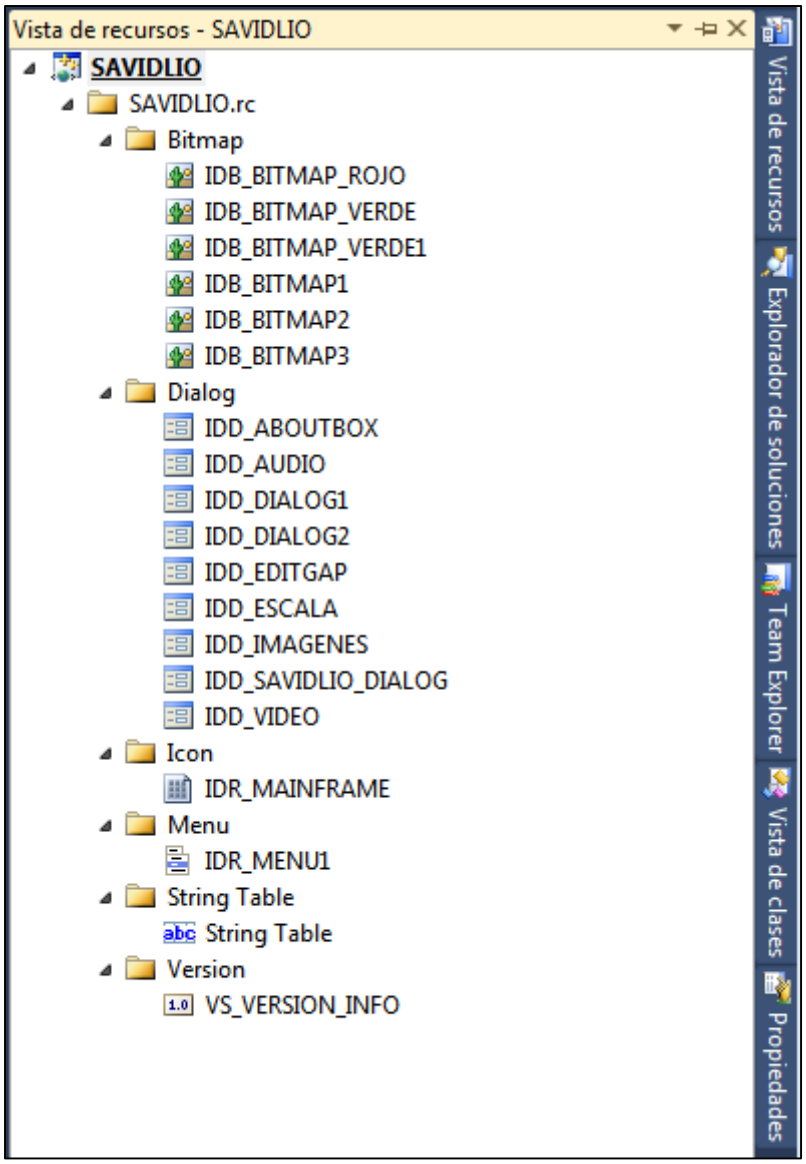


Ilustración 33: vista de recursos

Capa: indica la capa a la que pertenece la clase dentro de la arquitectura de tres campas en la que se ha estructurado la aplicación.

Referencias: indica las directivas de preprocesador que hacen referencia a otras clases de la aplicación.

Métodos y funciones: informa los nombres de los métodos y funciones, junto a una descripción de su funcionalidad.

Las clases de la aplicación SAVIDLIO son las siguientes:

CONTENEDOR PRINCIPAL: a continuación, se describe la clase del contenedor principal dónde se aloja el menú y las pestañas informativas de los sectores de vídeo auxiliar, audio auxiliar, *edit gap* e imágenes.

Encabezado	SAVIDLIODlg.h
CPP	SAVIDLIODlg.cpp
Recursos	IDD_SAVIDLIO_DIALOG
Capa	IU
Referencias	#include "stdafx.h" #include "SAVIDLIO.h" #include "SAVIDLIODlg.h" #include "afxdialogex.h" #include "SAVIDLIO_VIDEOAUX.h" #include "SAVIDLIO_AUDIOAUX.h" #include "SAVIDLIO_BL.h" #include "SAVIDLIO_DAL.h" #include "SAVIDLIO_DAL_EDITGAP.h" #include "SAVIDLIOTOOL.h" #include "VariablesGlobales.h"
Métodos y funciones	**DoDataExchange:** es ejecutado por el marco de trabajo en la carga inicial. En este cuadro de diálogo no cumple ninguna funcionalidad pero se mantiene al haber sido creado por el *wizard*. **OnInitDialog:** es ejecutado por el marco de trabajo en la carga inicial. Inserta las pestañas principales que componen la ventana y el menú de sistema. **OnArchivoAbrir:** es ejecutado al pulsar sobre el ítem "Archivo/Abrir" del menú principal. Ejecuta SAVIDLIO Procesador creando los ficheros binarios que son analizados. **OnAnalizar:** es ejecutado al pulsar sobre el ítem "Analizar" del menú principal. Se llama al método OnInitDialog() que actualiza las pestañas de vídeo auxiliar y audio auxiliar.

	OnReportesV32779: es ejecutado al pulsar sobre el ítem "Reportes/ Vídeo/Audio" del menú principal. Cumple con la funcionalidad de crear y abrir el reporte de resultados del análisis de los sectores de vídeo auxiliar y audio auxiliar. **OnEditGapReport:** es ejecutado al pulsar sobre el ítem "Reportes/ Edit Gap" del menú principal. Cumple con la funcionalidad de crear y abrir el reporte de resultados del análisis de las áreaas de relleno, *edit gap*. **OnQueryDragIcon:** es ejecutado por el marco de trabajo al arrastrar la ventana minimizada. Es creado por el *wizard*. **OnTcnSelchangeTab1:** es ejecutado al cambiar de pestaña. No cumple una funcionalidad específica en este proyecto pero hay que mantenerlo para no producir un error en el sistema. **InfoPorcentajesReport:** es llamado desde el método OnEditGapReport(). Calcula el porcentaje de coincidencia para cada uno de los marcadores que componen los reportes *edit gap*, en función del cuadro, pista y número de *edit gap*. **OnAyudaMarcoprincipal:** es ejecutado al pulsar sobre el ítem "Ayuda/Marco Principal" del menú principal. Permite abrir la guía de usuario del marco principal. **OnAyudaAudioauxiliar:** es ejecutado al pulsar sobre el ítem "Ayuda/Audio Auxiliar" del menú principal. Permite abrir la guía de usuario de audio auxiliar. **OnAyudaEditgap:** es ejecutado al pulsar sobre el ítem "Ayuda/*Edit Gap*" del menú principal. Permite abrir la guía de usuario de *edit gap*. **OnAyudaVideoauxiliar:** es ejecutado al pulsar sobre el ítem "Ayuda/Vídeo Auxiliar" del menú principal. Permite abrir la guía de usuario de vídeo auxiliar.

VÍDEO AUXILIAR (PESTAÑA): a continuación, se describe la clase de la pestaña que muestra los resultados del análisis de los metadatos del sector de vídeo auxiliar.

Encabezado	SAVIDLIO_VIDEOAUX.h
CPP	SAVIDLIO_VIDEOAUX.cpp
Recursos	IDD_VIDEO
Capa	IU
Referencias	#include "SAVIDLIO.h" #include "SAVIDLIO_VIDEOAUX.h" #include "SAVIDLIO_BL.h" #include "SAVIDLIO_DAL.h" #include "VariablesGlobales.h" #include "SAVIDLIOdlgCode.h" #include "SAVIDLIOTOOL.h" #include "stdafx.h" #include "afxdialogex.h"
Métodos y funciones	**DoDataExchange:** cumple la funcionalidad de actualizar los campos informativos de la pestaña. Es ejecutado en la carga inicial por el marco de trabajo o tras modificarse algún campo de interacción aprovechando que se ejecuta cada vez que se llama al método OnInitDialog(). **OnBnClickedButtonVaux:** es ejecutado al pulsar sobre el botón "CÓDIGO". Abre el cuadro de diálogo del código de Vídeo Auxiliar. **OnBnClickedRadio1:** es ejecutado al seleccionar el *radio button* "CUADRO 1". Actualiza la pantalla con los datos del primer cuadro. **OnBnClickedRadio2:** es ejecutado al seleccionar el *radio button* "CUADRO 2". Actualiza la pantalla con los datos del segundo cuadro. **OnBnClickedCheckModeCode:** es ejecutado cuando se marca el *check box* "CODE MODE". Actualiza la pantalla al llamar al método OnInitDialog().

	vista: es llamado desde DoDataExchange, escondiendo o visualizando los campos que informan el código binario, en función del valor de la variable “modo”. El valor del parámetro “modo” es asignado en el método OnBnClickedCheckModeCode(). **OnBnClickedButtonAyuda:** es ejecutado al pulsar sobre el botón “?”. Permite abrir la guía de usuario de vídeo auxiliar.

AUDIO AUXILIAR: a continuación, se describe la clase de la pestaña que muestra los resultados del análisis de los metadatos del sector de audio auxiliar.

Encabezado	SAVIDLIO_AUDIOAUX.h
Clase	SAVIDLIO_AUDIOAUX.cpp
Recursos	IDD_AUDIO
Capa	IU
Referencias	#include "SAVIDLIO.h" #include "SAVIDLIO_AUDIOAUX.h" #include "SAVIDLIO_BL.h" #include "SAVIDLIO_DAL.h" #include "VariablesGlobales.h" #include "SAVIDLIOdlgCodeAAUX.h" #include "SAVIDLIOTOOL.h" #include "stdafx.h" #include "afxdialogex.h"
Métodos y funciones	**DoDataExchange:** cumple la funcionalidad de actualizar los campos informativos de la pestaña. Es ejecutado en la carga inicial por el marco de trabajo o tras modificarse algún campo de interacción, aprovechando que se ejecuta cada vez que se llama al método OnInitDialog(). **OnBnClickedButtonAaux:** es ejecutado al pulsar sobre el botón “CÓDIGO”. Abre el cuadro de diálogo del código de Audio Auxiliar.

	OnBnClickedRadio1: es ejecutado al seleccionar el *radio button* "CUADRO 1". Actualiza la pantalla con los datos del primer cuadro. **OnBnClickedRadio2:** es ejecutado al seleccionar el *radio button* "CUADRO 2". Actualiza la pantalla con los datos del segundo cuadro. **OnBnClickedCheckModeCode:** es ejecutado al marcar el *check box* "CODE MODE". Actualiza la pantalla al llamar al método OnInitDialog(). **Vista:** es llamado desde DoDataExchange, escondiendo o visualizando los campos que informan el código binario, en función del valor de la variable "modo". El valor del parámetro "modo" es asignado en el método OnBnClickedCheckModeCode(). **OnBnClickedButtonAyuda:** es ejecutado al pulsar sobre el botón "?". Permite abrir la guía de usuario de audio auxiliar.

EDIT GAP: a continuación, se describe la clase de la pestaña que muestra los resultados del análisis de los datos de relleno, *edit gap*.

Encabezado	SAVIDLIO_EDITGAP.h
Clase	SAVIDLIO_EDITGAP.cpp
Recursos	IDD_EDITGAP
Capa	IU
Referencias	#include "SAVIDLIO.h" #include "SAVIDLIO_EDITGAP.h" #include "SAVIDLIO_DAL_EDITGAP.h" #include "VariablesGlobales.h" #include "SAVIDLIO_Escala.h" #include "SAVIDLIOTOOL.h" #include "stdafx.h" #include "afxdialogex.h"

Métodos y funciones	**DoDataExchange:** cumple la funcionalidad de actualizar los campos informativos de la pestaña. Es ejecutado en la carga inicial por el marco de trabajo o tras modificarse algún campo de interacción aprovechando que se ejecuta cada vez que se llama al método OnInitDialog(). **OnRefrescarUno:** actualiza la sección de la pantalla correspondiente al cuadro 1, asignando y habilitando campos en función de las selecciones que el usuario haya hecho en los campos de interacción (*combo boxes* y *radio buttons*). **OnRefrescarDos:** actualiza la sección de la pantalla que corresponde al cuadro 2, asignando y habilitando campos en función de las selecciones que el usuario haya hecho en los campos de interacción (*combo boxes* y *radio buttons*). **OnBnClickedRadio1:** es ejecutado al seleccionar el *radio button* "CUADRO 1". Actualiza la pantalla con los datos del primer cuadro. **OnBnClickedRadio2:** es ejecutado al seleccionar el *radio button* "CUADRO 2". Actualiza la pantalla con los datos del segundo cuadro. **OnCbnSelchangeComboNEditgap:** es ejecutado al seleccionar un ítem en el *combo box* "Nº EDIT GAP". Actualiza la sección del cuadro 1 o 2 en función del *radio button* que este seleccionado. No se actualiza la gráfica de escala de color. **OnCbnSelchangeComboPatron:** es ejecutado al seleccionar un ítem en el *combo box* "PATRÓN". Actualiza la sección del cuadro 1 ó 2 en función del *radio button* que este seleccionado. No se actualizan las etiquetas informativas de porcentajes totales y numero de bits. **OnCbnSelchangeComboPista:** es ejecutado al seleccionar un ítem en el *combo box* "PISTA". Actualiza la sección del cuadro 1 ó 2 en función del *radio button* que este seleccionado. No se actualiza la gráfica de escala de color.

	OnBnClickedButton3: es ejecutado al pulsar sobre el botón "+". Pasa a visualizarse los siguientes 25 bits del cuadro 1. **OnBnClickedButton4:** es ejecutado al pulsar sobre el botón "-", pasando a visualizarse los anteriores 25 bits del cuadro 1. **OnBnClickedButton5:** es ejecutado al pulsar sobre el botón "+", pasando a visualizarse los siguientes 25 bits del cuadro 2. **OnBnClickedButton6:** es ejecutado al pulsar sobre el botón "-", pasando a visualizarse los anteriores 25 bits del cuadro 2. **ComparaBits:** compara los bits del *edit gap* seleccionado con sus respectivos bits del patrón seleccionado. Visualiza una etiqueta verde en caso de que ambos tengan el mismo valor binario o una etiqueta roja en caso contrario. **ActualizarComparacionUno:** por cada uno de los 25 campos del *edit gap* del campo 1 se hace una llama al método ComparaBits(), comparando su valor con el del campo patrón correspondiente. **ActualizarComparacionDos:** por cada uno de los 25 campos del *edit gap* del campo 2 se hace una llama al método ComparaBits(), comparando su valor con el del campo patrón correspondiente. **ActualizarLabelInfoUno:** actualiza la etiquetas informativas "% PATRÓN A" y "% PATRÓN B" de la sección del cuadro 1. **ActualizarLabelInfoDos:** actualiza la etiquetas informativas "% PATRÓN A" y "% PATRÓN B" de la sección del cuadro 2. **ImagenEscala1:** asigna el color de la primera sección del gráfico de porcentajes de coincidencia del cuadro 1. **ImagenEscala2:** asigna el color de la segunda sección del gráfico de porcentajes de coincidencia del cuadro 1. **ImagenEscala3:** asigna el color de la tercera sección del gráfico de porcentajes de coincidencia del cuadro 1. **ImagenEscala4:** asigna el color de la cuarta sección del gráfico de porcentajes de coincidencia del cuadro 1.

	ImagenEscala5: asigna el color de la quinta sección del gráfico de porcentajes de coincidencia del cuadro 1. **ImagenEscala1Dos:** asigna el color de la primera sección del gráfico de porcentajes de coincidencia del cuadro 2. **ImagenEscala2Dos:** asigna el color de la segunda sección del gráfico de porcentajes de coincidencia del cuadro 2. **ImagenEscala3Dos:** asigna el color de la tercera sección del gráfico de porcentajes de coincidencia del cuadro 2. **ImagenEscala4Dos:** asigna el color de la cuarta sección del gráfico de porcentajes de coincidencia del cuadro 2. **ImagenEscala5Dos:** asigna el color de la quinta sección del gráfico de porcentajes de coincidencia del cuadro 2. **ActualizarEtiquetasUno:** actualiza etiquetas del cuadro 1. **ActualizarEtiquetasDos:** actualiza etiquetas del cuadro 2. **OnBnClickedButtonAyuda:** es ejecutado al pulsar sobre el botón "?". Permite abrir la guía de usuario de *edit gap*.

IMÁGENES: a continuación, se describe la clase de la pestaña que muestra las imágenes de los dos cuadros, junto a los datos generales de la señal de vídeo y audio

Encabezado	SAVIDLIO_IMAGENES.h
Clase	SAVIDLIO_IMAGENES.cpp
Recursos	IDD_IMAGENES
Capa	IU

Referencias	#include "stdafx.h" #include "SAVIDLIO.h" #include "SAVIDLIO_IMAGENES.h" #include "afxdialogex.h" #include "SAVIDLIOTOOL.h" #include "VariablesGlobales.h" #include "SAVIDLIO_BL.h"
Métodos y funciones	**DoDataExchange:** cumple la funcionalidad de actualizar los campos informativos de la pestaña. Es ejecutado en la carga inicial insertando las imágenes correspondientes a los cuadros y alimentando los campos informativos de la señal de vídeo y audio

CÓDIGO VÍDEO AUXILIAR (CUADRO DE DIÁLOGO): a continuación, se describe la clase del cuadro de diálogo que muestra el código contenido en el sector de vídeo auxiliar.

Encabezado	SAVIDLIOdlgCode.h
CPP	SAVIDLIOdlgCode.cpp
Recursos	IDD_DIALOG1
Capa	IU
Referencias	#include "stdafx.h" #include "SAVIDLIO.h" #include "SAVIDLIOdlgCode.h" #include "afxdialogex.h" #include "VariablesGlobales.h" #include "SAVIDLIO_DAL.h" #include "SAVIDLIO_BL.h" #include "SAVIDLIOTOOL.h" #include <windows.h>
Métodos y funciones	**DoDataExchange:** es ejecutado por el marco de trabajo en la carga inicial. En este cuadro de diálogo no cumple ninguna funcionalidad pero se mantiene al haber sido creado por el *wizard*.

	OnInitDialog: es ejecutado por el marco de trabajo en la carga inicial. Añade los ítems a los combos “CUADRO” y “PISTA” e inserta la imagen que informa si el formato del código cumple con la norma. **OnRefrescar:** permite actualizar la ventana cuando se produce una modificación en alguno de los dos *combos box,* informando los campos con los datos analizados de la pista y el cuadro seleccionados. **OnCbnSelchangeComboPista:** es ejecutado al seleccionar un ítem en el *combo box* “PISTA”. Actualiza la ventana llamando al método OnRefrescar(). **OnCbnSelchangeComboCuadro:** es ejecutado al seleccionar un ítem en el *combo box* “CUADRO”. Actualiza la ventana llamando al método OnRefrescar(). **OnBnClickedButtonAyuda:** es ejecutado al pulsar sobre el botón “AYUDA”. Permite abrir la guía de usuario del cuadro de diálogo del código de vídeo auxiliar.

CÓDIGO AUDIO AUXILIAR (CUADRO DE DIÁLOGO): a continuación, se describe la clase del cuadro de diálogo que muestra el código contenido en el sector de audio auxiliar.

Encabezado	SAVIDLIOdlgCodeAAUX.h
CPP	SAVIDLIOdlgCodeAAUX.cpp
Recursos	IDD_DIALOG2
Capa	IU

Referencias	#include "stdafx.h" #include "SAVIDLIO.h" #include "SAVIDLIOdlgCodeAAUX.h" #include "afxdialogex.h" #include "VariablesGlobales.h" #include "SAVIDLIOTOOL.h" #include "SAVIDLIO_DAL.h" #include "SAVIDLIO_BL.h"
Métodos y funciones	**DoDataExchange:** es ejecutado por el marco de trabajo en la carga inicial. En este cuadro de diálogo no cumple ninguna funcionalidad pero se mantiene al haber sido creado por el *wizard*. **OnInitDialog:** es ejecutado por el marco de trabajo en la carga inicial. Añade los ítems a los combos "CUADRO" y "PISTA" e inserta la imagen que informa si el formato del código cumple con la norma. **OnRefrescar:** permite actualizar la ventana cuando se produce una modificación en alguno de los dos *combos box,* informando los campos con los datos analizados de la pista y el cuadro seleccionados. **OnCbnSelchangeComboPista:** es ejecutado al seleccionar un ítem en el *combo box* "PISTA". Actualiza la ventana llamando al método OnRefrescar(). **OnCbnSelchangeComboCuadro:** es ejecutado al seleccionar un ítem en el *combo box* "CUADRO". Actualiza la ventana llamando al método OnRefrescar(). **OnBnClickedButtonAyuda:** es ejecutado al pulsar sobre el botón "AYUDA". Permite abrir la guía de usuario del cuadro de diálogo del código de audio auxiliar.

ESCALA (CUADRO DE DIÁLOGO): a continuación, se describe la clase del cuadro de diálogo que muestra la escala de color.

Encabezado	SAVIDLIOdlgCodeAAUX.h
CPP	SAVIDLIOdlgCodeAAUX.cpp
Recursos	IDD_DIALOG2
Capa	IU
Referencias	#include "stdafx.h" #include "SAVIDLIO.h" #include "SAVIDLIO_Escala.h" #include "afxdialogex.h" #include "VariablesGlobales.h" #include "SAVIDLIOTOOL.h"
Métodos y funciones	**DoDataExchange:** es ejecutado por el marco de trabajo en la carga inicial. Inserta la imagen en la ventana.

BL: a continuación, se describe la clase que contiene el código que implementa la capa de negocio.

Encabezado	SAVIDLIO_BL.h
CPP	SAVIDLIO_BL.cpp
Recursos	N/A
Capa	BL
Referencias	#include "StdAfx.h" #include "VariablesGlobales.h" #include "SAVIDLIO_BL.h" #include "SAVIDLIO_DAL.h"
Métodos y funciones	**analizarVideoAuxBin:** es la función que retorna la interpretación del código binario de cada uno de los campos del vídeo auxiliar.

	videoAuxBin: es la función que retorna el código binario de cada campo del vídeo auxiliar. **analizarAudioAuxBin:** es la función que retorna la interpretación del código binario de cada uno de los campos del audio auxiliar. **audioAuxBin:** es la función que retorna el código binario de cada campo del audio auxiliar. **analizarVideoAuxBinReport:** es la función que retorna la interpretación del código binario de cada uno de los campos del vídeo auxiliar. Es usado para el caso específico de los reportes. **analizarAudioAuxBinReport:** es la función que retorna la interpretación del código binario de cada uno de los campos del audio auxiliar. Es usado para el caso específico de los reportes.

DAL: a continuación, se describe la clase que contiene el código que implementa la capa de acceso a datos.

Encabezado	SAVIDLIO_DAL.h
CPP	SAVIDLIO_DAL.cpp
Recursos	N/A
Capa	DAL
Referencias	#include "StdAfx.h" #include "VariablesGlobales.h" #include "SAVIDLIO_DAL.h" #include "SAVIDLIOTOOL.h" #include <afx.h>
Métodos y funciones	**extraerContCode:** retorna la secuencia del código hexadecimal insertado en el campo que muestra el código total, en el cuadro de diálogo del código de vídeo auxiliar y audio auxiliar. El tamaño de la secuencia es igual al que se define en la norma para las filas que componen el sector.

	extraerContHex: retorna todo el código hexadecimal contenido en el fichero, del sector, pista y cuadro introducidos como parámetros. **extraerVideoAuxBin:** es la función que retorna el código binario del campo introducido como parámetro del sector de vídeo auxiliar. **extraerVideoAuxHex:** es la función que retorna el código hexadecimal de *video source* o *video source control*. **extraerAudioAuxBin:** es la función que retorna el código binario del campo introducido como parámetro del sector de audio auxiliar. **extraerAudioAuxHex:** es la función que retorna el código hexadecimal de *audio source* o *audio source control*.

DAL (EDIT GAP): a continuación, se describe la clase que contiene el código que implementa la capa de acceso a datos de *edit gap*.

Encabezado	SAVIDLIO_DAL_EDITGAP.h
CPP	SAVIDLIO_DAL_EDITGAP.cpp
Recursos	N/A
Capa	DAL
Referencias	#include "StdAfx.h" #include "SAVIDLIO_DAL_EDITGAP.h" #include "VariablesGlobales.h" #include "SAVIDLIOTOOL.h"
Métodos y funciones	**extaerEditGap:** extrae una parte del código binario del sector *edit gap* en función de los valores introducidos como parámetros.

HERRAMIENTAS: a continuación, se describe la clase que contiene las funciones comunes para su uso en cualquiera del resto de las clases.

Encabezado	SAVIDLIOTOOL.h
CPP	SAVIDLIOTOOL.cpp
Recursos	N/A
Capa	N/A
Referencias	#include "StdAfx.h" #include "SAVIDLIOTOOL.h" #include <iomanip> #include <conio.h> #include <stdlib.h> #include <iostream> #include <stdio.h>
Métodos y funciones	**getArchivo:** permite abrir y extraer el contenido del código hexadecimal del fichero cuyo nombre se introduce como parámetro. **conversorHB:** convierte el código hexadecimal introducido como parámetro en el código binario que es retornado. **rutaImagen:** devuelve la ruta donde se encuentra un archivo que es insertado o abierto por la aplicación. **ruta:** tiene la misma funcionalidad que la función “rutaimagen” pero devuelve un valor de tipo CString

VARIABLES GLOBALES: a continuación, se describe la clase donde se declaran y asignan las constantes globales de la aplicación.

Encabezado	VariablesGlobales.h
CPP	VariablesGlobales.cpp
Recursos	N/A
Capa	N/A
Referencias	#include "StdAfx.h" #include "VariablesGlobales.h"

11.4 IMPLEMENTACIÓN DE LA FUNCIONALIDAD DE *REPORTING*

Se describen en esta sección los pasos seguidos para implementar el sistema de gestión de reportes de la aplicación.

En todo el conjunto de clases que forman parte de MFC, no hay ninguna específica para la administración y gestión de reportes. Tras probar infructuosamente la integración con Crystal Report de .Net y tener la opción de configurar el proyecto para que sea compatible con CLR (*Common Languages Runtime*), que cuenta con C++ como uno de los lenguajes de programación, se ha optado por utilizar el *framework* .Net como solución al problema.

Como contrapartida, la compatibilidad con CLR inhabilita del IDE herramientas de ayuda como Itellisense (autocompletado), por lo que hasta no haber tenido terminado todo el desarrollo de la IU, no se ha configurado en este modo.

La clase del *framework* que se ha usado para implementar la solución es "Microsoft.Office.Iterop.Word", que permite manejar documentos Word y entre otras funcionalidades asignar valores a marcadores por medio de referencias a los nombres que lo identifican. Al no haber información de cómo utilizar esta clase con el lenguaje C++, ni siquiera en la ayuda MSDN de Microsoft, se tratará de explicar con detalle la configuración y la implementación para garantizar el futuro mantenimiento de la aplicación.

Sintetizando, la aplicación instanciará un objeto de un documento Word que sirve como plantilla y en la que se han insertado marcadores que serán manejados desde la aplicación para ser informados con los datos de la IU.

PLANTILLAS WORD:

Se han creado dos ficheros de extensión .docx que sirven como plantillas de los reportes:

- video_aud.docx
- edit_gap.docx

Las plantillas están estructuradas con tablas y ambas divididas diferenciando entre los cuadros uno y dos.

En la plantillas video_aud.docx es donde se reporta la información que contienen los metadatos de los sectores de vídeo y audio auxiliar, dividiéndose a su vez en las variantes Source y Source Control de cada uno de los sectores. Las tablas además de la cabecera dónde se identifican las columnas, contiene dos filas, la superior donde se indica la información interpretada de las palabras de código de los metadatos y la inferior donde se informa el código binario original que ha sido procesado.

La plantillas edit_gap.docx está dividida en tablas en función de los números de *Edit Gap* uno, dos y tres. Cada tabla contiene doce columnas correspondientes a las doce pistas que forman un cuadro y dos filas correspondientes a los patrones A y B. La información que contiene cada celda, es el porcentaje de coincidencia entre los bits de los *Edit Gap* y el patrón correspondiente.

Las celdas de cada tabla están en blanco de forma predeterminada y contienen un marcador que sirve de contendor para la información insertada desde SAVIDLIO.

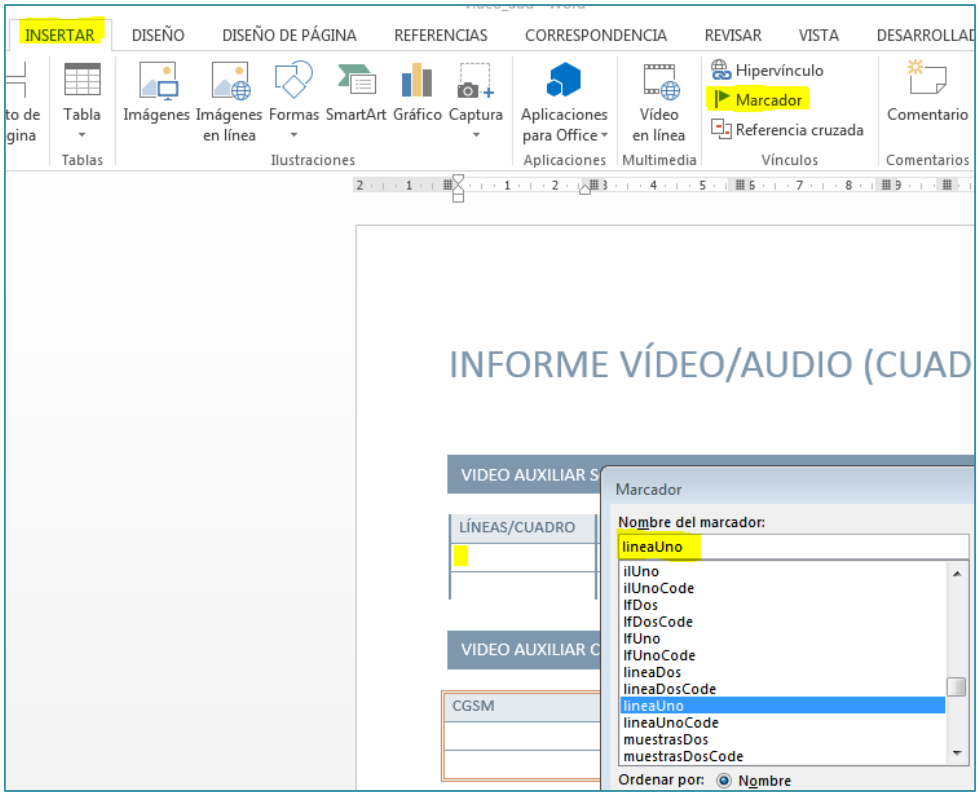


Ilustración 34: marcadores en documento Word

Se han creado 3 subcarpetas en la carpeta de ruta absoluta“//reportes”:

- **plantillas**: contiene los archivos originales que sirven de plantilla. Estos nunca son modificados, solo copiados a la carpeta “*temp*”, manteniendo siempre el formato original y evitando el borrado accidental de marcadores de la plantilla que provocaría una excepción del sistema.
- **temp**: contiene una copia de las plantillas y son los ficheros manipulados desde la aplicación.
- **read**: contiene una copia de los archivos temporales que son los que se ejecutan en Office Word. La función de esta copia es evitar la excepción provocada al volver a ejecutar el proceso de reporte teniendo abierto el fichero y tratando de salvarlo.

SOLUCIÓN SOFTWARE:

Como se comentó en el prólogo del apartado, se ha visto necesario configurar el proyecto para que sea compatible con CLR y así poder importar la clase “Microsoft.Office.Interop.Word” que permite manejar documentos Word.

Para configurarlo en modo compatibilidad con CLR y así con el *framework* .Net, se accederá siguiendo la ruta desde el menú principal, “Proyecto/Propiedades” o por el *hotkey* de acceso rápido “Alt + F7”.

En la ventana modal de configuración del proyecto, se selecciona “Propiedades de configuración/General” y dentro de la pestaña “Valores Predeterminados del Proyecto”, en la propiedad Compatibilidad con CLR, se ha pasado del valor predeterminado “No Compatible con Common Languages Runtime” al valor “Compatible con Common Languages Runtime (/clr)”.

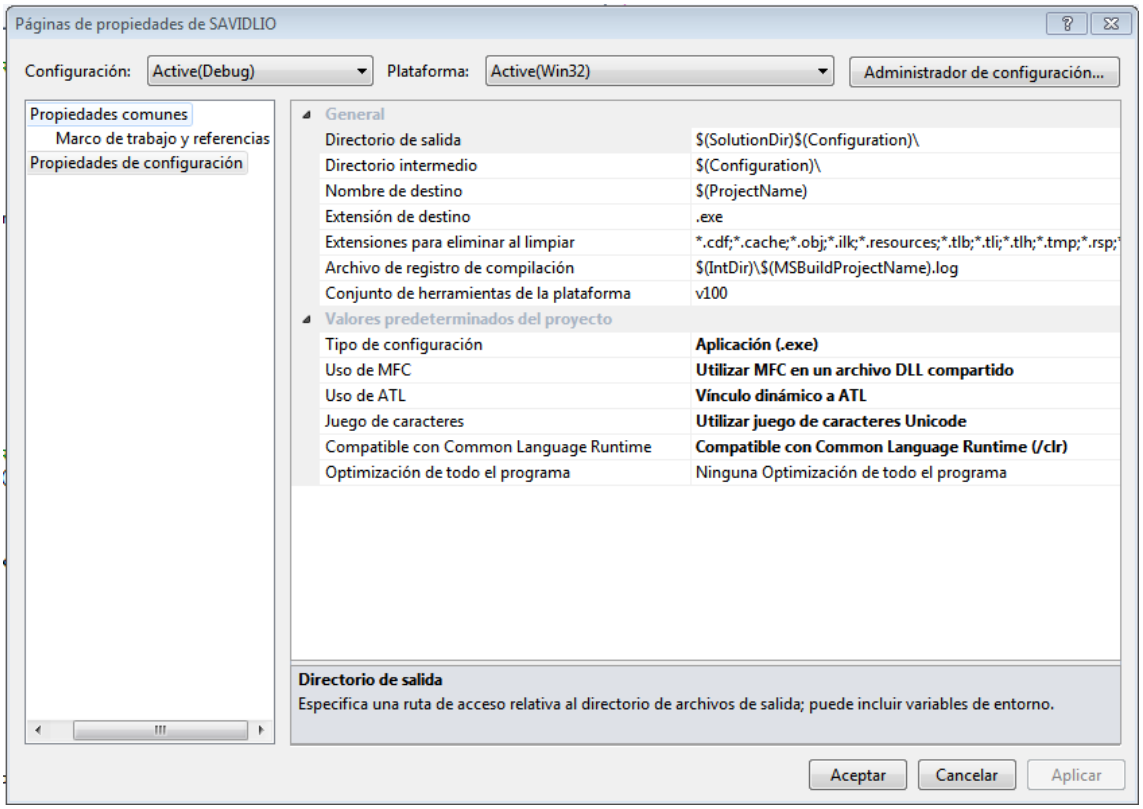


Ilustración 35: configuración de compatibilidad con CLR

Con esta configuración ya se pueden agregar referencias al proyecto, pulsando el botón derecho con el cursor sobre el proyecto raíz y haciendo clic sobre la opción “Referencias”.

Se abrirá la ventana de propiedades, visualizándose la opción “Propiedades Comunes”, donde se encuentran los botones para agregar y quitar referencias.

Se han agregado al proyecto las clases:

- “Microsoft.Office.Interop.Word”
- “System”

12. FASE DE PRUEBAS

12.1 OBJETIVOS Y CARACTERÍSTICAS DE LA FASE DE PRUEBAS

Cuando un desarrollador novel piensa en pruebas de *software* suele caer en dos errores habituales. Por un lado piensa que el objetivo final de una fase de pruebas es demostrar que el *software* funciona correctamente y, por otro, que la duración de la fase de pruebas no ocupa una parte significativa de tiempo dentro del ciclo de vida del *software*.

El objetivo principal de la fase de pruebas no es otro que la de reproducir errores para identificarlos y corregirlos, evaluando así la calidad del *software* y depurando el código. En caso de no encontrar errores, entonces sí se verifica que el funcionamiento del desarrollo es el esperado. Esta fase suele consumir una proporción aproximada del 30% respecto al total del ciclo de vida, por lo que su estudio y diseño requiere un interés especial a la hora de garantizar una planificación eficiente del proyecto.

Paradójicamente, mientras que en las fases de diseño y desarrollo los esfuerzos se centran en construir el programa, en la fase de pruebas el fin es “destruirlo”. La razón es que el usuario puede interaccionar con la herramienta de una manera inadecuada, por lo que en la fase de pruebas se trata de reproducir estas hipotéticas malas prácticas, por muy inverosímiles que nos parezcan, para ver en qué condiciones puede darse un error para corregirlo o una interrupción para capturarla.

Es importante diferenciar los conceptos de verificar, cuyo fin es constatar que el *software* funciona, y validar, que determina si el *software* desarrollado cumple con los requisitos especificados por el usuario. Para cumplir con los requisitos de verificación y validación se debe realizar un seguimiento de los requisitos del cliente durante la ejecución de las pruebas.

Una herramienta útil para este propósito son las matrices de trazabilidad que permiten relacionar las pruebas con los requisitos.

Para facilitar el desarrollo y mejorar la eficiencia, la fase de pruebas debe planificarse mucho antes de ser implementada en las fases más tempranas del desarrollo. Respecto a la ejecución de las pruebas, se realiza durante todo el ciclo de vida y no solo al finalizar la fase de desarrollo, siendo conveniente que este haya alcanzado cierto grado de madurez; en caso contrario, se puede incurrir en pérdidas de tiempo innecesarias. Es aconsejable también que el equipo que ejecute las pruebas sea independiente al equipo de desarrollo, ya que, al no estar familiarizado con el *software,* es más fácil que cometa prácticas sobre el mismo que reproduzcan errores.

Por último, para terminar esta breve descripción de alto nivel, nos preguntamos si es necesario tratar de llegar a dar una cobertura del cien por cien del código en las pruebas. En desarrollos complejos es imposible abarcar el cien por cien de cobertura, por lo que es aconsejable centrarla en las partes del código con más probabilidades de inducir errores. El Principio de Pareto es aplicable a este respecto, augurando que un 20% de todo el código contendrá el 80% de los errores.

En el apartado de Calidad de *software* se decidió seguir con la técnica TDD, por lo que vamos a garantizar una cobertura prácticamente del cien por cien y que las pruebas se van a hacer conjuntamente con el desarrollo.

12.2 CASOS DE PRUEBA

En el plan de pruebas se describen los casos de prueba que detallan las características de la ejecución. El estándar IEEE 610 define un caso de prueba como:

Conjunto de entrada de prueba, condición de ejecución, y resultado esperado desarrollados para un objetivo en particular, como el ejercicio de una ruta de programa en particular o para verificar el cumplimiento de un requisito específico, y como documentación que especifique las entradas, los resultados previos, y un conjunto de condiciones de ejecuciones de un elemento de prueba.

Se deduce del estándar que los casos de prueba tienen que cumplir con los siguientes requisitos:

- Describir los N parámetros de entrada
- Procedimiento a seguir en la ejecución
- Relacionar al menos un caso de prueba con cada uno de los casos de uso descritos en el EDRF
- Comparar los resultados esperados con los resultados obtenidos
- Ser específicos de un módulo del código
- Deben estar documentados

En algunas ocasiones, cuando algunas de las fases previas del ciclo de vida han sido mal planteadas, la fase de pruebas es la que más tiempo y esfuerzo conlleva. Por esta razón, un buen diseño de los casos de pruebas se convierte en una herramienta útil para compensar estas demoras.

12.3 DOCUMENTACIÓN EN LA FASE DE PRUEBAS

La fase de pruebas evoluciona en paralelo con el desarrollo y continúa una vez haya finalizado este. Para poder documentar toda la fase de pruebas se debe redactar un documento por cada una de las fases en las que se desarrolla.

Las partes en las que se estructura una fase de pruebas son las siguientes:

- Planificación de las pruebas
- Implementación de las pruebas
- Obtención de resultados de las pruebas
- Toma de decisiones sobre la base de los resultados (correcciones, replanteos, etc.)

Cada una de estas partes debe esta descrita en los documentos siguientes **[29]:**

- Planificación de las pruebas
    - Plan de pruebas

    - Especificación de diseño de pruebas
    - Especificación de casos de prueba
    - Especificación de procedimientos de prueba
    - Informe de transferencia de elementos de prueba
- Implementación de las pruebas
    - Registro de pruebas
    - Informe de incidentes
- Obtención de resultados de las pruebas y toma de decisiones en función de los resultados (correcciones, replanteos, etc.)
    - Informe de resumen de pruebas

Se tratará de adaptar el citado estándar a las necesidades específicas del proyecto. A continuación se desglosa la descripción de los documentos según son especificados en el estándar para, posteriormente, justificar la selección de los documentos que se redactarán como anexos del proyecto.

PLANIFICACIÓN DE LAS PRUEBAS

En esta fase se especifica qué, cómo y cuándo se va a probar, por lo que la documentación debería describir los métodos que se aplicarán, los tipos de pruebas, las herramientas y el calendario de pruebas, entre otros.

Plan de pruebas:

Es, tal vez, el documento más importante, en el cual se detallan las directrices que se seguirán durante la fase de las pruebas. El plan de pruebas tiene que contener la siguiente información sobre:

- Actores partícipes en las pruebas
- Recursos humanos y herramientas de *software* necesarias
- Se especifican los módulos, funcionalidades y componentes que van a ser probados, junto con las metodologías a aplicar
- Calendario de planificación de las pruebas
- Cobertura y expectativas de resultados de salida

El detalle del plan de pruebas debe contener los siguientes campos:

- Identificador del plan de pruebas: debe permitir identificar el plan de pruebas de cualquier otro plan de forma única, además de indicar el nivel (versión del *software*) y *software* en el que se ejecuta. Este puede ser la concatenación del nombre del *software*, la versión, la fecha y el autor
- Referencias/requerimientos: se indican referencias a otros planes, al igual que los casos de uso que son probados en el plan de pruebas
- Introducción: es un breve resumen ejecutivo en el que se indica cuál es el alcance del plan en relación con el proyecto *software*. Puede incluir información de diferente naturaleza, como son los recursos, la cualificación de ejecutor, las herramientas, etc.
- Elementos de prueba: es la lista de los módulos que van a ser probados y la relación que tienen estos con los requisitos. Pueden incluir referencias a incidentes, solicitudes de mejora, etc.
- *Software* crítico: es donde se identifican los puntos que se consideran críticos a la hora de realizar las pruebas
- Características que deben ser probadas: es donde se detallan las características de lo que se ha probado con vistas a que el usuario esté informado, por lo que no debe estar escrito en un lenguaje técnico
- Riesgo: es el nivel de riesgo de cada elemento de prueba, que sirve para valorarlo en una escala que puede ser H, M o L —alto medio o bajo— siendo comprensible por el usuario
- Características que no se analizarán/probarán: es la lista de las funciones que no se van a probar justificando cuáles son las razones
- Enfoque: es donde se indica cuál es la estrategia a seguir en el plan de pruebas
- Criterios de inicio y fallo: es donde se indica cuál es el criterio que es considerado para determinar que el plan ha terminado cumpliendo con su objetivo.
- Criterios de suspensión y reanudación de los requisitos: indicar cuáles pueden ser las razones para dejar pausado un plan de pruebas o poner fin a una serie de pruebas. También indica si un plan es pausado o cuáles son los impactos potenciales
- Resultados/entrega de la prueba: indica qué documentación se va a entregar como informe de resultados del plan de pruebas

- Tareas de prueba: debe incluir las dependencias y los niveles de habilidad del ejecutor
- Necesidades de entorno: se detallan algunos requisitos extraordinarios que puedan ser necesarios, como puede ser un *hardware*, *software*, uso restringido del sistema durante la prueba, etc.
- *Staffing* y necesidades de formación: se identifican los perfiles especiales de los que vayan a ejecutar las pruebas o la formación que va a ser necesaria en caso de usar una herramienta que las automatice
- Responsabilidades: son los roles que van a adoptar cada uno de los participantes durante todo el periodo que involucre la fase de pruebas
- Planificación: debe basarse en mayor medida en la realidad, por lo que si se desvía la estimación del desarrollo también lo hará la fase de pruebas; por tanto, estará condicionada a otras fases del ciclo de vida del *software*. Se marcarán los hitos de toda la fase de pruebas
- Riesgos y contingencias: se identifican riesgos como falta de *staff*, la falta de cualificación, posibles retrasos de entrega, cambios en los requisitos y el plan de contingencia asociado con tal de minimizar las consecuencias
- Aprobaciones: debemos conocer quién o quiénes serán las personas que pueden aprobar los procesos y permitir proceder con el siguiente nivel. Se distinguirán los niveles técnicos de los de negocio
- Glosario: permite un entendimiento común de todas las partes involucradas en la fase de pruebas

Documento de especificación de diseño de pruebas

En este documento se especifican las funcionalidades que se quieren probar, evaluando cuáles son las que aportan mayor valor e indicando cuáles son las condiciones en las que se considera el resultado como un éxito o un fracaso. Este documento, pues, sirve, básicamente, para priorizar y focalizar esfuerzos en aquellos módulos que requieren un funcionamiento óptimo y, así, no malgastar el tiempo en casos de prueba que no lo requieran.

Documento de especificación de casos de prueba

Tras el análisis previo en el que se han categorizado por relevancia las funcionalidades, se describen con detalle las pruebas por funcionalidad con los casos de prueba.

Los datos que describen un caso de prueba son los siguientes:

• Parámetros de entrada (argumentos) que se usarán para realizar la prueba

• Resultado esperado tras realizar la prueba

• Precondiciones

• Relaciones entre casos de prueba

Documento de especificación de los procedimientos de prueba

En este documento se detallan los pasos a seguir de cómo se realiza la prueba, cómo debe estar configurado el entorno y el orden en el que se van a realizar.

Informe de transmisión de elementos de pruebas

Es un documento donde se describen los elementos que aplican en una prueba, permitiendo su rápida localización. En muchos casos el creador de las pruebas no es el ejecutor de las mismas, por lo que este texto puede aportar mucho valor.

DOCUMENTACIÓN DURANTE LA EJECUCIÓN DE LAS PRUEBAS

Mientras se realizan las pruebas hay que tener unos documentos que nos sirvan para realizar el seguimiento de las mismas, informándonos de su estado y de las incidencias encontradas.

Registro de pruebas

Este documento es donde registran las pruebas y sus resultados los encargados de ejecutarlas. Un objetivo fundamental del proceso de *testing* es proporcionar información acerca del sistema que se está probando.

Este documento debe tener información de los siguientes puntos:

- Identificador de la prueba
- Persona encargada de ejecutarla
- Fecha
- Orden en que se han realizado las pruebas
- Versión sobre la que se realiza (trazabilidad)

- Resultado de verificación de la prueba

Informe de incidentes

Este documento es donde se detallan las posibles discrepancias entre los resultados obtenidos y los esperados. No solo debe tener información para identificar el incidente, sino que también debe contener información para su resolución.

Datos del documento:

- Identificador de la prueba en que se ha identificado el incidente
- Fase de la prueba donde se ha reproducido
- Parámetros de entrada y de salida
- Resultados esperados
- Entorno
- Valoración del impacto
- Propuestas para su resolución

DOCUMENTACIÓN PARA LA FINALIZACIÓN DE LAS PRUEBAS

Informe de resumen de pruebas

Cuando los encargados de realizar las pruebas han superado un nivel de pruebas, tienen que reportar los resultados tanto a los creadores como a aquellos actores encargados de decidir cómo proceder. Este informe contiene los siguientes puntos:

- Pruebas realizadas
- Tiempo consumido
- Valoraciones de calidad del proceso de pruebas
- Nivel de calidad alcanzado en el *software*

12.4 DESCRIPCIÓN DEL TIPO DE PRUEBAS

Una vez se ha determinado cómo proceder con la documentación necesaria para planificar, registrar y reportar resultados de la fase de pruebas y antes de justificar el plan que se seguirá en este proyecto, nos queda por describir qué tipo de pruebas están estandarizadas.

Niveles de pruebas

Pruebas de código: durante el desarrollo, el desarrollador debe comprobar que no hay errores de compilación ni de ejecución en el código. Es una actividad constante que se realiza cada vez que se depura o compila el proyecto. Hay técnicas que permiten reducir el tiempo que estas ocupan durante el desarrollo, como por ejemplo el "*attachament*", y que serán practicadas durante el desarrollo.

Pruebas unitarias [30]: se diseñan y ejecutan con el propósito de garantizar el correcto funcionamiento de uno de los módulos en particular de todos los que componen el *software,* sin requerir para su implementación que la totalidad del *software* esté desarrollado. El objetivo principal de este tipo de pruebas es mejorar aspectos referentes a la calidad del *software*, disminuyendo los tiempos de corrección y depuración. Por el contrario, se suele consumir mucho tiempo especificándolas y planificándolas.

Para un diseño óptimo, se busca cumplir con las siguientes características:

- Automatización, evitando la intervención manual
- Alta cobertura, probando la máxima cuantía de líneas de código
- Replicación, permitiendo su repetición tantas veces como sea necesario
- Minimización de tiempos de ejecución; como consecuencia de la replicación, el tiempo que estas tardan en ejecutarse es importante a la hora de diseñarlas
- Independencia, evitando solapamientos entre casos de prueba
- Integridad del entorno, revirtiendo, una vez finalizada la ejecución, al estado previo de las condiciones del entorno

Por otro lado, las pruebas son de gran ayuda a la hora de comprender lo que se espera de un módulo, motivo por el que aportan información auxiliar para la redacción de la documentación final.

Para garantizar el correcto funcionamiento de todo el sistema es necesario complementarlas con pruebas de integración y pruebas del sistema, detalladas a continuación.

Pruebas de integración [31]: tienen como objetivo evaluar el funcionamiento conjunto de un grupo de elementos unitarios que componen un proceso. Se ejecutan con posterioridad a las pruebas unitarias, garantizando la correcta integración entre las interfaces de los módulos en los que se estructura el *software*. Como complemento a estas, cuando se han finalizado se ejecutan las pruebas de sistema.

En este tipo de pruebas, se pueden diferenciar las siguientes variables:

-Integración Incremental:

Se van probando los módulos combinados con aquellos que ya han sido probados de una forma incremental, hasta llegar a testar el *software* en su totalidad. Hay dos maneras de realizar este tipo de pruebas, que son la integración incremental ascendente y la integración incremental descendente.

- Integración incremental ascendente:

1. En este tipo de pruebas se parte de los módulos de más bajo nivel, agrupando aquellos que cumplan con una función o subfunción específica.

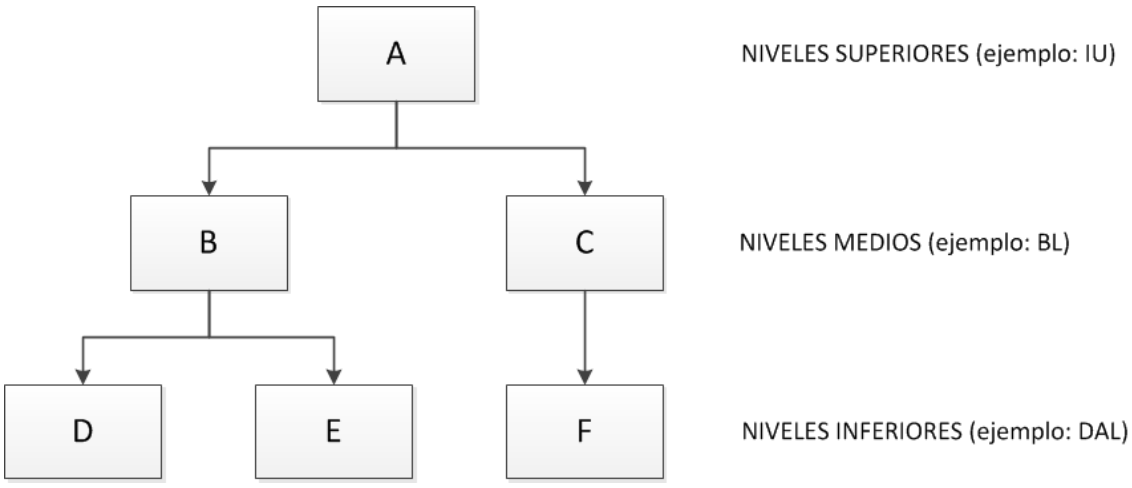


Ilustración 36: integración incremental ascendente (paso 1)

2. Simulamos con un *driver*—módulo conductor— por módulo, las llamadas a otros módulos, la introducción de parámetros como argumentos y la obtención de resultados.

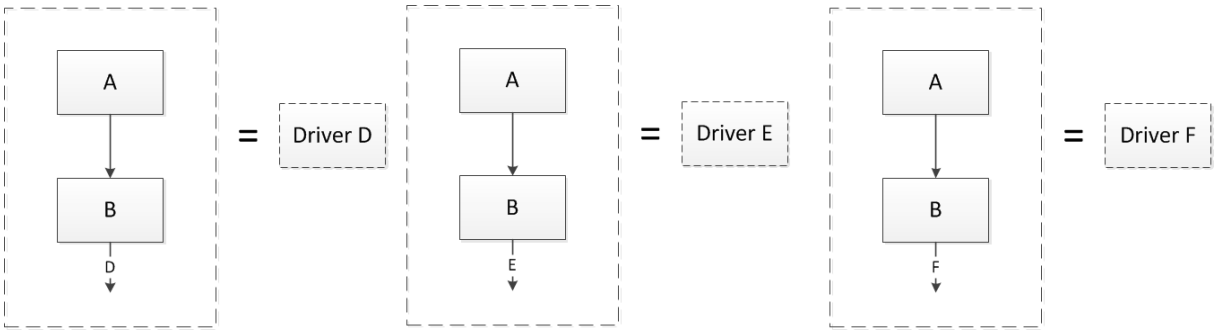


Ilustración 37: integración incremental ascendente (paso 2)

3. Se realiza la prueba de cada módulo por medio de su *driver*.

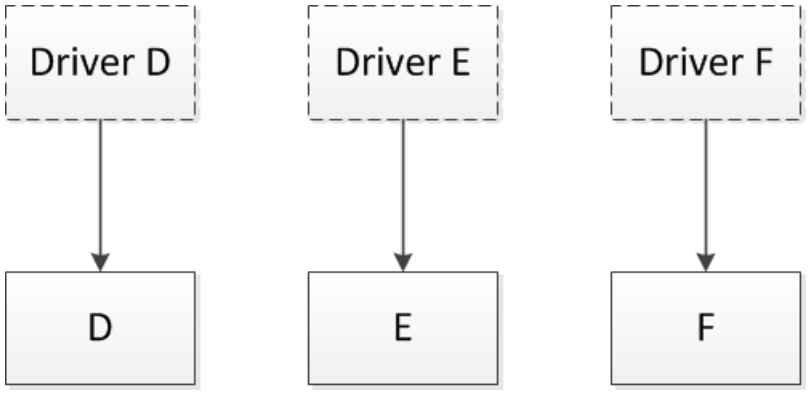


Ilustración 38: integración incremental ascendente (paso 3)

4. Se sustituyen los *driver* por los módulos superiores con sus entradas y salidas reales.

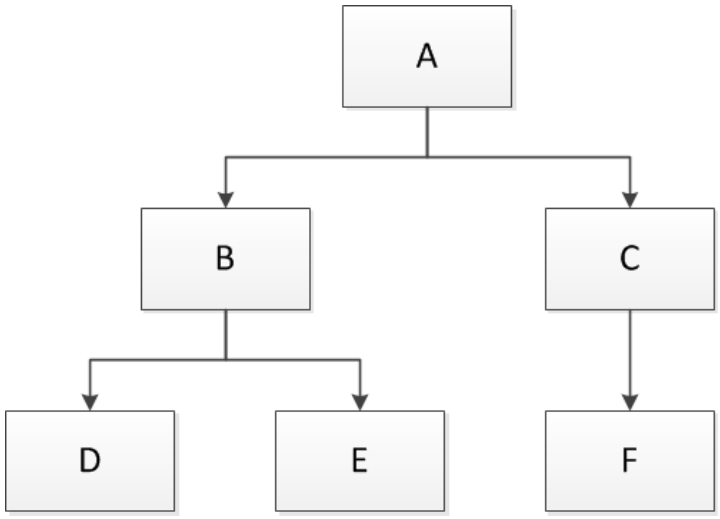


Ilustración 39: integración incremental ascendente (paso 4)

Por este motivo, siempre se empezará probando los módulos que recuperan los datos de las BBDD o, en nuestro caso, de los archivos de texto para terminar probando los módulos que componen la interfaz de usuario.

Las ventajas que presentan este tipo de pruebas es que, en los niveles inferiores, al cumplir con las funcionalidades más específicas, las entradas son más fáciles de diseñar y, al ser en estas donde se obtienen los resultados, es más fácil prever qué valores cumplirán con lo estimado.

Al alimentarse el resto de niveles de los datos recuperados por aquellos más inferiores, permite que, cuanto más se avance en el incremento, el número de errores descienda y sea más fácil realizar las pruebas.

La gran desventaja es que hay que codificar los *drivers* que simularán las llamadas de los módulos superiores.

-Integración descendente:

Al contrario de la integración ascendente, se comienza por los módulos de mayor nivel hasta llegar, de forma progresiva, a los módulos de niveles inferiores. La regla para desarrollar este tipo de pruebas es que para incorporar módulos de niveles inferiores, antes deben haberse probado los módulos que lo llaman.

Hay dos formas de llevar a cabo esta integración:

- Se van integrando los módulos en horizontal con módulos del mismo nivel
- Se van integrando en vertical hasta cubrir una rama

Etapas:

- El módulo de nivel superior es el que hace de *driver*, codificando módulos ficticios que simulan los niveles inferiores
- Probar de manera reiterativa cada vez que se sustituye un módulo ficticio por el original que lo suplanta

Las ventajas principales son que los fallos de los módulos inferiores se detectan de una forma temprana y permiten ver la estructura general desde un principio.

La principal desventaja de este tipo de integración es que hay que codificar un gran número de módulos ficticios y heterogéneos. Otra desventaja es que los juegos de datos son difíciles de obtener, ya que los suelen generar los niveles inferiores.

-Integración no incremental:

Es, tal vez, la más adecuada para el presente proyecto, ya que es de sencilla aplicación y da buenos resultados en pequeños proyectos. Consiste en probar cada módulo de forma independiente para terminar probándolos en conjunto como un todo.

Así, evitaremos la tediosa tarea de tener que codificar *drivers* o módulos ficticios aunque nos encontramos con el inconveniente de que hasta que finalice la prueba conjunta no sabremos el alcance y el volumen de los errores.

Pruebas de sistema [32]: son pruebas que testean el *software* en su totalidad y van más allá del proceso de desarrollo del mismo. Suelen ser pruebas de caja negra sin que el ejecutor de las pruebas tenga la necesidad de acceder al código y tampoco que este sea el ingeniero del *software*.

Una vez el *software* esté concluido, verificado y validado por el usuario, se realizan estas pruebas para comprobar la integración del nuevo *software* con el resto de elementos del sistema.

Con estas pruebas se trata de evaluar los siguientes aspectos:

- Rendimiento: se mide por el tiempo que tarda la aplicación en responder ante una acción realizada por el usuario. Depende de muchos factores, como el tamaño de los datos que se transmiten y el canal, en caso de aplicaciones que se comuniquen con BBDD, o de la calidad de arquitectura del *software*
- Carga: llamadas pruebas de *stress*, tratan de poner al límite la aplicación para delimitar cuál es su nivel de resistencia y acotar las condiciones en las que se garantiza un funcionamiento óptimo sin que la aplicación se colapse
- Robustez: cuantifica la capacidad de la aplicación de soportar entradas incorrectas
- Seguridad: trata de auditar las formas de acceder a la aplicación y cuáles son los roles de los usuarios, definiendo los privilegios de accesibilidad
- Usabilidad: califica la facilidad que presenta el usuario a la hora de interaccionar con la aplicación
- Instalación**:** determina el grado de complejidad en la instalación del *software,* como los requisitos del sistema mínimos que soporta
- Pruebas de regresión: son pruebas realizadas con posterioridad a una modificación del código de un *software*, debida a un correctivo o a un cambio de alcance, y que garantizan la integridad de las funcionalidades previas a la modificación
- Pruebas funcionales: son las pruebas que tienen el fin de validar si el *software* cumple con los requisitos y, por lo tanto, con la funcionalidad especificada por el usuario
- Caja blanca [32]: son un tipo de pruebas de *software* que se realizan sobre las funciones internas de un módulo. Así como las pruebas de caja negra ejercitan los requisitos funcionales desde el exterior del módulo, las de caja blanca están dirigidas a las funciones internas

Método de prueba de camino básico:

Las pruebas de caja blanca se realizan con el objetivo de verificar el código, tratando de dar el máximo de cobertura posible. Para dar un cien por cien de cobertura de un módulo es necesario, al menos una vez, pasar por todos los caminos.

Con el fin de cumplir con esto se utiliza una herramienta muy útil, llamada grafos de flujo, que permite calcular el número ciclomático que nos da el número de juego de datos que necesitamos si queremos ejecutar todo el código.

V(G) = E-N + 2 Siendo E el número de aristas del flujo dibujado y N los nodos.

Es posible calcularla como:

- V(G) = N+1 (N ramas cerradas)
- V(G)= N+1 (N número de instrucciones no secuenciales) Estructuras de control condicional y bucles

El número ciclomático nos da el número de caminos independientes que se pueden dar. Como camino independiente entendemos el flujo de instrucciones de programa que introduce, por lo menos, un nuevo conjunto de sentencias de procesamiento y se asociará un caso de prueba para cada camino independiente.

Caja negra [31]: son las pruebas que validan el correcto funcionamiento de las interfaces y que el resultado obtenido es el que se esperaba. También sirven para evaluar el rendimiento del sistema. No necesitan tener acceso al código, por lo que pueden ser ejecutadas por los propios usuarios dando una visión externa de la funcionalidad de la aplicación; son complementarias de las pruebas de caja blanca. A continuación se describen algunas de las variantes de este tipo de pruebas:

-Pruebas de tabla ortogonal: son útiles en aplicaciones que tienen un número pequeño de parámetros de entrada, teniendo cada parámetro un número determinado de valores. Esto permite que se puedan hacer pruebas exhaustivas con cada uno de los valores de entrada, introduciéndolos en tablas y relacionándolos con los resultados de salida esperados.

-Pruebas de comparación: la idea de este tipo de pruebas es construir una serie de prototipos que reproducen la funcionalidad externa de la aplicación a probar. Introduciendo los mismos parámetros deben obtenerse los mismos resultados para que la prueba dé un resultado conforme. Estos prototipos tienen que ser programados con tecnologías de fácil codificación.

-Análisis de los Valores Límite (AVL): es el método utilizado para evaluar y verificar las condiciones límite, siendo necesario complementarlas con pruebas de comparación.

Estadísticamente, la gran mayoría de los errores se concentra en los valores límite, por lo que estas pruebas se basan en asignar los valores de prueba justo en los valores correlativos a ambos lados del límite.

-Método basado en grafos:

Para la ejecución de este método es necesario conocer los objetos que se modelan en la aplicación y las relaciones entre ellos, dibujando grafos de objetos y las relaciones entre sí, diseñando la serie de pruebas en base a los diagramas.

12.5 DISEÑO DE LA FASE DE PRUEBAS EN SAVID

Los objetivos principales que se van a perseguir en la fase de pruebas de la herramienta SAVID son:

- Buscar errores y excepciones que den resultados incorrectos o interrupciones durante la ejecución
- Comprobar el buen funcionamiento de los diferentes módulos y su correcta integración
- Garantizar una comunicación correcta entre el sistema y el resto de sistemas que componen el entorno
- En caso de realizar algún evolutivo en un componente, comprobar que no se introducen errores en otros componentes
- Validar con el tutor que se cumple con la funcionalidad especificada

La planificación de las pruebas a alto nivel se dará en el siguiente orden:

Se comienza con el diseño del plan de pruebas antes de comenzar el desarrollo. La implementación de las pruebas se realiza de manera incremental, desde el detalle de código hasta el *software* en su conjunto, de manera que durante el desarrollo se ejecutarán las pruebas de código, teniendo en cuenta técnicas que ahorren tiempo en las depuraciones. Una vez se hayan desarrollado módulos completos se podrán realizar pruebas unitarias y, una vez esté concluido el desarrollo, pruebas integrales, de sistema y de usuario. En caso de realizar evolutivos o modificaciones en el código, se realizarán pruebas de regresión para garantizar que no se han introducido errores en otros módulos. Tanto la documentación como el seguimiento y control se realizarán durante todo el ciclo de vida.

13. CONCLUSIONES

Aunque las conclusiones se han ido expresando en cada uno de los apartados, en éste se tratarán de sintetizar de una manera general.

El riesgo que se observó con la tecnología MFC, con la que se ha implementado la aplicación y que se definió como uno de los requisitos principales, finalmente ha sido uno de los inconvenientes principales con los que me he encontrado durante el desarrollo. Al no ser una tecnología enfocada para el desarrollo de cuadros de diálogo, que sería el tipo de aplicación idónea para el fin perseguido, los tipos de componentes gráficos son limitados y con funcionalidades básicas. Ciertas acciones como insertar imágenes, abrir archivos binarios o añadir información a marcadores de archivos Word, y que no deberían generar problemas de implementación, han incurrido en pérdidas de tiempo, teniendo incluso que generar métodos específicos para cumplir dichas funcionalidades. La falta de soporte de ayuda en la red, incluso en la web de ayuda al desarrollador MSDN (Microsoft Developer Network), ha sido otro hándicap con el que se contaba, cumpliéndose finalmente los augurios.

Otro de los puntos conflictivos ha sido a la hora de implementar la creación de reportes. La tecnología carece de clases específicas para este fin, por lo que finalmente se tuvo que optar por integrar el proyecto con CLR (Common Lenguage Runtime) y aplicar clases que cumpliesen con las funcionalidades necesarias.

Aún con todos los inconvenientes anteriores, no se ha optado por renunciar a ninguna de las funcionalidades y se ha conseguido implementar lo que en un principio se pensó de manera satisfactoria.

El seguir las metodologías ágiles para el desarrollo, no ha sido posible aplicarla en su totalidad. El dividir el desarrollo en sprint de duración de un mes, agrupando el desarrollo en funcionalidades, ha permitido la suficiente flexibilidad para que los requisitos y funcionalidades fueran mejorando conforme el software maduraba. Por lo que los prototipos planteados en un principio, han evolucionado a una aplicación más compleja funcionalmente.

Las reuniones de principio y fin de sprint no han sido posibles realizarlas, por falta de disponibilidad tanto mías como del tutor. La automatización de pruebas, otro de los puntos importantes de esta metodología no ha sido implementado por no aportar valor y por las pérdidas de tiempo debidas al aprendizaje necesario para configurar las aplicaciones que realizan estas automatizaciones.

Por lo que se concluye que estas metodologías aportan un gran valor para desarrollos de este tipo, aunque no sean tan fáciles de aplicar como en un principio parecen.

1. ÍNDICE DE ILUSTRACIONES

2. BIBLIOGRAFÍA

1. DV, de Wikipedia, julio 2013, [Online] http://en.wikipedia.org/wiki/DV/. **Pág. 16**.

2. Luís I. Ortiz Berenguer y José Luis Rodríguez Vázquez, *Ingeniería de Vídeo en Entornos Digitales*, "Capítulo 9: Registro de la Señal de Vídeo Digital en Cinta" Madrid: Ediciones Universidad Politécnica de Madrid, diciembre 2008. **Pág. 17**.

3. Codificación Digital Video (DV, DVCAM, DVCPRO), de Sostenibilidad de los formatos digitales planificación de la Biblioteca del Congreso Colecciones, Abril 2013, [Online] http://www.digitalpreservation.gov/. **Pág. 18**.

4. "Soportes de grabación de vídeo (Apuntes)", Departamento de Ingeniería Audiovisual y Comunicaciones, Universidad Politécnica de Madrid, diciembre 2010. **Pág. 21**.

5. "Software Product Evaluation", ISO/IEC 9126, 1991. **Pág. 35**.

6. Introducción a TDD (Test Driven Development), Oriol del Barrio, abril 2011, [Online] http://blog.lordudun.es/2011/04/introduccion-a-tdd-test-driven-development. **Pág. 36**.

7. Carlos Blé Jurado*, Diseño Ágil con TDD*, Primera Edición iExpertos, enero de 2010]. **Pág. 37**.

8. Dan North, *Introducing BDD*, revista Better Software, marzo 2006]. **Pág. 49**.

9. Buenas Prácticas: La Integración Continua, Luis Fraile, Diciembre 2011, [Online] http://www.genbetadev.com/metodologias-de-programacion/buenas-practicas-la-integracion-continua. **Pág. 40**.

10. James Shore, *The Art of Agile Development*, USA: Editorial O'Reilly, octubre 2007. **Pág. 41**.

11. Colaboradores de Wikipedia. Revisión de código, de Wikipedia. [en línea]. *Wikipedia, la enciclopedia libre*. [Fecha de consulta: marzo de 2013]. Disponible en: http://es.wikipedia.org/wiki/Revisión_de_código. **Pág. 41**.

12. CIOAL. Análisis estático: El Eslabón Perdido de la Calidad del Software [en línea]. CIO América Latina. [Fecha de consulta: agosto de 2010]. Disponible en: http://www.cioal.com/2010/08/18/analisis-estatico-el-eslabon-perdido-de-la-calidad-del-software/. **Pág. 42**.

13. Refactorización, de Wikipedia, agosto 2010, [Online] http://es.wikipedia.org/wiki/Refactorización. **Pág. 42**.

14. Sobre convenciones y notaciones (húngara, CamelCase, etc), Camilo Sperberg, marzo 2011, [Online] http://blog.unreal4u.com/2011/03/sobre-convenciones-y-notaciones-hungara-camelcase-etc/. **Pág. 42**.

15. Standard for Television Digital Recording - 6.35-mm Type D-7, SMPTE 306M, septiembre 1998. **Pág. 48**.

16. Curso: Metodologías ágiles SCRUM, Departamento PMO, Everis Spain, febrero 2012. **Pág. 64**.

17. Information Technology – Software life cycle processes, ISO/IEC 12207, 1995. **Pág. 66**.

18. Desarrollo en cascada, de Wikipedia, febrero 2013, [Online]
http://es.wikipedia.org/wiki/Desarrollo_en_cascada. **Pág. 69**.

19. Laboratorio Nacional de Calidad de Software, *Ingeniería del Software: Metodologías y Ciclos de Vida*, Ministerio de Industria, Turismo y Comercio, marzo 2009. **Pág. 70**.

20. Desarrollo iterativo y creciente, de Wikipedia, marzo 2013, [Online]
http://es.wikipedia.org/wiki/Desarrollo_iterativo_y_creciente. **Pág. 71**.

21. Modelo de prototipos, de Wikipedia, enero 2013, [Online]
http://es.wikipedia.org/wiki/Modelo_de_prototipos. **Pág. 72**.

22. Curso: Metodologías ágiles SCRUM, Departamento PMO, Everis Spain, Febrero 2012. **Pág. 72**.

23. Miguel Vega, *Casos de Uso UML*, Universidad de Granada, octubre 2010. **Pág. 76**.

24. Ignacio García-Caro García, PFC: Aplicación web para el conocimiento y conversión de unidades, UNED, enero 2010. **Pág. 78**.

25. Patricio Salinas Caro, *Tutorial: UML*, Departamento de Ciencias de la Computación, Universidad de Chile, marzo 2004. **Pág. 78**.

26. Dr. Víctor Braberman, *Ingeniería de Software*, Departamento de Computación, Universidad de Buenos Aires, febrero 2005. **Pág. 79**.

27. Jesús Cáceres Tello, *Diagramas de Secuencia*, Departamento Ciencias de la Computación, Universidad de Alcalá, febrero 2013. **Pág. 81**.

28. Arquitectura documento/vista, MSDN Microsof, [Online] http://msdn.microsoft.com/es-es/library/cc485517(v=vs.71).aspx. **Pág. 91**.

29. Standard for Software and System Test Documentation, IEEE-829, 2008. **Pág. 116**.

30. Curso: Validación y Verificación de Software, Departamento *Utilities*, Everis Spain, enero 2013. **Pág. 122**.

31. Grupo Alarcos, *Calidad de los Sistemas de Información,* Escuela Superior de Informática de Ciudad Real. **Págs 123 y 128**.

32. Pruebas de Software “Testing”, Miguel Ángel Barrera, Agosto 2010, [Online]
http://85517atesting.blogspot.com.es/2010/08/pruebas-de-software-testing.html. **Págs. 126 y 127**.

ANEXOS